

A

B

C

D

1

2

3

4

5

—○ GND

—○ D0

—○ D1

—○ D2

—○ D3

—○ D4 MCM6830A

—○ D5 ROM

—○ D6 Socket

—○ D7

—○ CS0

—○ CS1'

—○ Vcc

A0 ○—

A1 ○—

A2 ○—

A3 ○—

A4 ○—

A5 ○—

A6 ○—

A7 ○—

A8 ○—

A9 ○—

CS3 ○—

CS2 ○—

—○ A7

—○ A6

—○ A5

—○ A4

—○ A3

—○ A2

—○ A1 2716 EPROM

—○ A0

—○ O0

—○ O1

—○ O2

—○ GND

Vcc ○—

A8 ○—

A9 ○—

Vpp ○—

G' ○—

A10 ○—

E'P ○—

O7 ○—

O6 ○—

O5 ○—

O4 ○—

O3 ○—

**20**

1

2

4

5

6

9

10

12

13

8

A

B

C

D





# Individual Learning Program

## MICROPROCESSORS

### *Unit 1*

## NUMBER SYSTEMS AND CODES

EE-3401

HEATH COMPANY  
BENTON HARBOR, MICHIGAN 49022  
595-2039-02

Copyright © 1977  
Heath Company  
All Rights Reserved  
Printed in the United States of America

## CONTENTS

Introduction .....	1-3
Unit Objectives .....	1-4
Unit Activity Guide .....	1-5
Decimal Number System .....	1-6
Binary Number System .....	1-11
Octal Number System .....	1-20
Hexadecimal Number System .....	1-29
Binary Codes .....	1-42
Experiments 1 and 2 .....	1-58
Unit Examination .....	1-59
Examination Answers .....	1-61



## *Unit 1*

# NUMBER SYSTEMS AND CODES

## INTRODUCTION

The purpose of this first unit on microprocessors is to give you a firm foundation in number systems and codes. Binary numbers and codes are the basic language of all microprocessors. Octal and hexadecimal numbers allow easy manipulation of binary numbers and data. Thus, a good foundation in numbers and codes is essential to understanding microprocessors.

This unit will reacquaint you with the decimal number system, then expand the basic concept of numbers to the binary, octal, and hexadecimal systems. Understanding these systems fully will help you understand the many digital codes used with microprocessors. Although this unit can only give you a working knowledge of numbers and codes, you will become more proficient with them as you proceed through the units that follow.

A listing of number system tables has been provided in Appendix B of this course. Appendix B is located at the back of the second binder.

Examine the Unit Objectives listed in the next section to see what you will learn in this unit. Then follow the instructions in the Unit Activity Guide to be sure you perform all of the steps necessary to complete this lesson successfully. Check off each step as you complete it and, in the spaces provided, keep track of the time you spend on each activity.

## UNIT OBJECTIVES

When you complete this unit you will have the following knowledge and capabilities:

1. Given any decimal number, you will be able to convert it into its binary, octal, hexadecimal, and BCD equivalent.
2. Given any binary number, you will be able to convert it into its decimal, octal, hexadecimal, and BCD equivalent.
3. Given any octal number, you will be able to convert it into its decimal and binary equivalent.
4. Given any hexadecimal number, you will be able to convert it into its decimal and binary equivalent.
5. Given any BCD code, you will be able to convert it into its decimal and binary equivalent.
6. Given a list of popular digital codes, you will be able to read and identify them including pure binary, natural 8421 BCD, Gray, ASCII, and BAUDOT.
7. You will be able to convert a letter or number into its ASCII binary code, and convert an ASCII binary code into its letter or number equivalent.
8. You will be able to define the following terms:

Radix	BCD
Integer	Gray Code
Decimal	ASCII
Binary	BAUDOT
Octal	Most Significant Bit (MSB)
Hexadecimal	Least Significant Bit (LSB)
Bit	Most Significant Digit (MSD)
Parity	Least Significant Digit (LSD)

## UNIT ACTIVITY GUIDE

Completion  
Time

- |   |       |
|---|-------|
| <input type="checkbox"/> Read section on Decimal Number System.     | _____ |
| <input type="checkbox"/> Answer Self-Test Review questions 1 — 6.   | _____ |
| <input type="checkbox"/> Read section on Binary Number System.      | _____ |
| <input type="checkbox"/> Answer Self-Test Review questions 7 — 13.  | _____ |
| <input type="checkbox"/> Read section on Octal Number System.       | _____ |
| <input type="checkbox"/> Answer Self-Test Review questions 14 — 19. | _____ |
| <input type="checkbox"/> Read section on Hexadecimal Number System. | _____ |
| <input type="checkbox"/> Answer Self-Test Review questions 20 — 25. | _____ |
| <input type="checkbox"/> Read section on Binary Codes.              | _____ |
| <input type="checkbox"/> Answer Self-Test Review questions 26 — 36. | _____ |
| <input type="checkbox"/> Perform Experiments 1 and 2.               | _____ |
| <input type="checkbox"/> Complete the Unit Examination.             | _____ |
| <input type="checkbox"/> Check Examination Answers.                 | _____ |

## DECIMAL NUMBER SYSTEM

The number system we are all familiar with is the decimal number system. This system was originally devised by Hindu mathematicians in India about 400 A.D. The Arabs began to use the system about 800 A.D., where it became known as the Arabic Number System. After it was introduced to the European community about 1200 A.D., the system soon acquired the title "decimal number system."

A basic distinguishing feature of a number system is its **base** or **radix**. The base indicates the number of characters or digits used to represent quantities in that number system. The decimal number system has a base or radix of 10 because we use the ten digits 0 through 9 to represent quantities. When a number system is used where the base is not known, a subscript is used to show the base. For example, the number  $4603_{10}$  is derived from a number system with a base of 10.

**Positional Notation** The decimal number system is positional or weighted. This means each digit position in a number carries a particular weight which determines the magnitude of that number. Each position has a weight determined by some power of the number system base, in this case 10. The positional weights are  $10^0$  (units)\*,  $10^1$  (tens),  $10^2$  (hundreds), etc. Refer to Figure 1-1 for a condensed listing of powers of 10.

$10^0$	= 1
$10^1$	= 10
$10^2$	= 100
$10^3$	= 1,000
$10^4$	= 10,000
$10^5$	= 100,000
$10^6$	= 1,000,000
$10^7$	= 10,000,000
$10^8$	= 100,000,000
$10^9$	= 1,000,000,000

Figure 1-1

Condensed listing of powers of 10.

\*Any number with an exponent of zero is equal to one.

We evaluate the total quantity of a number by considering the specific digits and the weights of their positions. For example, the decimal number 4603 is written in the shorthand notation with which we are all familiar. This number can also be expressed with positional notation.

$$\begin{aligned}(4 \times 10^3) + (6 \times 10^2) + (0 \times 10^1) + (3 \times 10^0) &= \\(4 \times 1000) + (6 \times 100) + (0 \times 10) + (3 \times 1) &= \\4000 + 600 + 0 + 3 &= 4603_{10}\end{aligned}$$

To determine the value of a number, multiply each digit by the weight of its position and add the results.

**Fractional Numbers** So far, only **integer** or whole numbers have been discussed. An integer is any of the natural numbers, the negatives of these numbers, or zero (that is, 0, 1, 4, 7, etc.). Thus, an integer represents a whole or complete number. But, it is often necessary to express quantities in terms of fractional parts of a whole number.

Decimal fractions are numbers whose positions have weights that are **negative powers of ten** such as  $10^{-1} = \frac{1}{10} = 0.1$ ,  $10^{-2} = \frac{1}{100} = 0.01$ , etc.

Figure 1-2 provides a condensed listing of negative powers of 10 (decimal fractions).

$$10^{-1} = \frac{1}{10} = 0.1$$

$$10^{-2} = \frac{1}{100} = 0.01$$

$$10^{-3} = \frac{1}{1000} = 0.001$$

$$10^{-4} = \frac{1}{10,000} = 0.0001$$

$$10^{-5} = \frac{1}{100,000} = 0.00001$$

$$10^{-6} = \frac{1}{1,000,000} = 0.000001$$

Figure 1-2  
Condensed listing of negative  
powers of 10.

A radix point (decimal point for base 10 numbers) **separates** the **integer** and **fractional** parts of a number. The integer or whole portion is to the left of the decimal point and has positional weights of units, tens, hundreds, etc. The fractional part of the number is to the right of the decimal point and has positional weights of tenths, hundredths, thousandths, etc. To illustrate this, the decimal number 278.94 can be written with positional notation as shown below.

$$\begin{aligned}(2 \times 10^2) + (7 \times 10^1) + (8 \times 10^0) + (9 \times 10^{-1}) + (4 \times 10^{-2}) &= \\(2 \times 100) + (7 \times 10) + (8 \times 1) + (9 \times 1/10) + (4 \times 1/100) &= \\200 + 70 + 8 + 0.9 + 0.04 &= 278.94_{10}\end{aligned}$$

In this example, the left-most digit ( $2 \times 10^2$ ) is the **most significant digit** or MSD because it carries the greatest weight in determining the value of the number. The right-most digit, called the **least significant digit** or LSD, has the lowest weight in determining the value of the number. Therefore, as the term implies, the MSD is the digit that will affect the greatest change when its value is altered. The LSD has the smallest effect on the complete number value.

## Self-Test Review

1. The \_\_\_\_\_ indicates the number of characters or digits in a number system.
2. In the decimal number system, the base or radix is \_\_\_\_\_.
3. Write the following numbers using positional notation.
  - A.  $4563_{10}$
  - B.  $26.32_{10}$
  - C.  $536.9_{10}$
4. In the decimal number system, the radix point is called the \_\_\_\_\_.
5. Convert the following positional notations into their shorthand decimal form.
  - A.  $(5 \times 10^1) + (2 \times 10^0) + (3 \times 10^{-1}) + (8 \times 10^{-2})$
  - B.  $(4 \times 10^{-1}) + (6 \times 10^{-2}) + (2 \times 10^{-3})$
  - C.  $(3 \times 10^3) + (7 \times 10^2) + (1 \times 10^1) + (0 \times 10^0)$
6. The radix point separates the \_\_\_\_\_ and \_\_\_\_\_ parts of a number.

## Answers

1. base or radix.
2. 10.
3. A.  $4563_{10} = 4000 + 500 + 60 + 3$ .  
 $= (4 \times 10^3) + (5 \times 10^2) + (6 \times 10^1) + (3 \times 10^0)$   
B.  $(2 \times 10^1) + (6 \times 10^0) + (3 \times 10^{-1}) + (2 \times 10^{-2})$   
C.  $(5 \times 10^2) + (3 \times 10^1) + (6 \times 10^0) + (9 \times 10^{-1})$
4. decimal point.
5. A.  $(5 \times 10^1) + (2 \times 10^0) + (3 \times 10^{-1}) + (8 \times 10^{-2}) =$   
 $(5 \times 10) + (2 \times 1) + (3 \times 1/10) + (8 \times 1/100) =$   
 $50 + 2 + 0.3 + 0.08 = 52.38_{10}$   
B.  $0.462_{10}$   
C.  $3710_{10}$
6. integer or whole, fractional.



## BINARY NUMBER SYSTEM

The simplest number system that uses positional notation is the binary number system. As the name implies, a **binary** system contains only two elements or states. In a number system this is expressed as a base of 2, using the digits 0 and 1. These two digits have the same basic value as 0 and 1 in the decimal number system.

Because of its simplicity, microprocessors use the binary number system to manipulate data. Binary data is represented by binary digits called **bits**. The term bit is derived from the contraction of **binary digit**. Microprocessors operate on groups of bits which are referred to as words. The binary number 11101101 contains eight bits.

### Positional Notation

As with the decimal number system, each bit (digit) position of a binary number carries a particular weight which determines the magnitude of that number. The weight of each position is determined by some power of the number system base (in this example 2). To evaluate the total quantity of a number, consider the specific bits and the weights of their positions. (Refer to Figure 1-3 for a condensed listing of powers of 2.) For example, the binary number 110101 can be written with positional notation as follows:

$$(1 \times 2^5) + (1 \times 2^4) + (0 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0)$$

To determine the decimal value of the binary number 110101, multiply each bit by its positional weight and add the results.

$$(1 \times 32) + (1 \times 16) + (0 \times 8) + (1 \times 4) + (0 \times 2) + (1 \times 1) = \\ 32 + 16 + 0 + 4 + 0 + 1 = 53_{10}$$

$2^0 = 1_{10}$	$2^6 = 64_{10}$
$2^1 = 2_{10}$	$2^7 = 128_{10}$
$2^2 = 4_{10}$	$2^8 = 256_{10}$
$2^3 = 8_{10}$	$2^9 = 512_{10}$
$2^4 = 16_{10}$	$2^{10} = 1024_{10}$
$2^5 = 32_{10}$	$2^{11} = 2048_{10}$

Figure 1-3  
Condensed listing of powers of 2.

Fractional binary numbers are expressed as negative powers of 2. Figure 1-4 provides a condensed listing of negative powers of 2. In positional notation, the binary number 0.1101 can be expressed as follows:

$$(1 \times 2^{-1}) + (1 \times 2^{-2}) + (0 \times 2^{-3}) + (1 \times 2^{-4})$$

To determine the decimal value of the binary number 0.1101, multiply each bit by its positional weight and add the results.

$$(1 \times 1/2) + (1 \times 1/4) + (0 \times 1/8) + (1 \times 1/16) = \\ 0.5 + 0.25 + 0 + 0.0625 = 0.8125_{10}$$

In the binary number system, the radix point is called the binary point.

$$2^{-1} = \frac{1}{2} = 0.5_{10}$$

$$2^{-2} = \frac{1}{4} = 0.25_{10}$$

$$2^{-3} = \frac{1}{8} = 0.125_{10}$$

$$2^{-4} = \frac{1}{16} = 0.0625_{10}$$

$$2^{-5} = \frac{1}{32} = 0.03125_{10}$$

$$2^{-6} = \frac{1}{64} = 0.015625_{10}$$

$$2^{-7} = \frac{1}{128} = 0.0078125_{10}$$

$$2^{-8} = \frac{1}{256} = 0.00390625_{10}$$


Figure 1-4  
Condensed listing of negative  
powers of 2.

## Converting Between the Binary and Decimal Number Systems

In working with microprocessors, you will often need to determine the decimal value of binary numbers. In addition, you will find it necessary to convert a specific decimal number into its binary equivalent. The following information shows how such conversions are accomplished.

**Binary to Decimal** To convert a binary number into its decimal equivalent, add together the weights of the positions in the number where binary 1's occur. The weights of the integer and fractional positions are indicated below.

INTEGER								FRACTIONAL		
$2^7$	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$	$2^{-1}$	$2^{-2}$	$2^{-3}$
128	64	32	16	8	4	2	1	.5	.25	.125

Binary Point 

As an example, convert the binary number 1010 into its decimal equivalent. Since no binary point is shown, the number is assumed to be an integer number, where the binary point is to the right of the number. The right-most bit, called the **least significant bit** or LSB, has the lowest integer weight of  $2^0 = 1$ . The left-most bit is the **most significant bit** (MSB) because it carries the greatest weight in determining the value of the number. In this example, it has a weight of  $2^3 = 8$ . To evaluate the number, add together the weights of the positions where binary 1's appear. In this example, 1's occur in the  $2^3$  and  $2^1$  positions. The decimal equivalent is ten.

Binary Number	1	0	1	0					
Position Weights	$2^3$	$2^2$	$2^1$	$2^0$					
Decimal Equivalent	8	+	0	+	2	+	0	=	$10_{10}$

To further illustrate this process, convert the binary number 101101.11 into its decimal equivalent.

Binary Number	1	0	1	1	0	1	.1	1									
Position Weights	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$	$2^{-1}$	$2^{-2}$									
Decimal Equivalent	32	+	0	+	8	+	4	+	0	+	1	+	.5	+	.25	=	$45.75_{10}$

**Decimal to Binary** A decimal integer number can be converted to a different base or radix through successive divisions by the desired base. To convert a decimal integer number to its binary equivalent, successively divide the number by 2 and note the remainders. When you divide by 2, the remainder will always be 1 or 0.

The remainders form the equivalent binary number.

As an example, the decimal number 25 is converted into its binary equivalent.

$25 \div 2 = 12$	with remainder 1	← LSB
$12 \div 2 = 6$		0
$6 \div 2 = 3$		0
$3 \div 2 = 1$		1
$1 \div 2 = 0$		1 ← MSB

Divide the decimal number by 2 and note the remainder. Then divide the quotient by 2 and again note the remainder. Then divide the quotient by 2 and again note the remainder. Continue this division process until 0 results. Then collect remainders beginning with the last or most significant bit (MSB) and proceed to the first or least significant bit (LSB). The number  $11001_2 = 25_{10}$ . Notice that the remainders are collected in the reverse order. That is, the first remainder becomes the least significant bit, while the last remainder becomes the most significant bit.

**NOTE:** Do not attempt to use a calculator to perform this conversion. It would only supply you with confusing results.

To further illustrate this, the decimal number 175 is converted into its binary equivalent.

$175 \div 2 = 87$	with remainder 1	← LSB
$87 \div 2 = 43$		1
$43 \div 2 = 21$		1
$21 \div 2 = 10$		1
$10 \div 2 = 5$		0
$5 \div 2 = 2$		1
$2 \div 2 = 1$		0
$1 \div 2 = 0$		1 ← MSB

The division process continues until 0 results. The remainders are collected to produce the number  $10101111_2 = 175_{10}$ .

To convert a decimal fraction to a different base or radix, multiply the fraction successively by the desired base and record any integers produced by the multiplication as an overflow. For example, to convert the decimal fraction 0.3125 into its binary equivalent, multiply repeatedly by 2.

$0.3125 \times 2 = 0.625 = 0.625$	with overflow	0	← MSB
$0.6250 \times 2 = 1.250 = 0.250$		1	
$0.2500 \times 2 = 0.500 = 0.500$		0	
$0.5000 \times 2 = 1.000 = 0$		1	← LSB

These multiplications will result in numbers with a 1 or 0 in the units position (the position to the left of the decimal point). By recording the value of the units position, you can construct the equivalent binary fraction. This units position value is called the "overflow." Therefore, when 0.3125 is multiplied by 2, the overflow is 0. This becomes the most significant bit (MSB) of the binary equivalent fraction. Then 0.625 is multiplied by 2. Since the product is 1.25, the overflow is 1. When there is an overflow of 1, it is effectively subtracted from the product when the value is recorded. Therefore, only 0.25 is multiplied by 2 in the next multiplication process. This method continues until an overflow with no fraction results. It is important to note that you can not always obtain 0 when you multiply by 2. Therefore, you should only continue the conversion process to the accuracy or precision you desire. Collect the conversion overflows beginning at the radix (binary) point with the MSB and proceed to the LSB. This is the same order in which the overflows were produced. The number  $0.0101_2 = 0.3125_{10}$ .

To further illustrate this process, the decimal fraction 0.90625 is converted into its binary equivalent.

$0.90625 \times 2 = 1.8125 = 0.8125$	with overflow	1	← MSB
$0.81250 \times 2 = 1.6250 = 0.6250$		1	
$0.62500 \times 2 = 1.2500 = 0.2500$		1	
$0.25000 \times 2 = 0.5000 = 0.5000$		0	
$0.50000 \times 2 = 1.0000 = 0$		1	← LSB

The multiplication process continues until either 0 or the desired precision is obtained. The overflows are then collected beginning with the MSB at the binary (radix) point and proceeding to the LSB. The number  $0.11101_2 = 0.90625_{10}$ .

If the decimal number contains both an integer and fraction, you must separate the integer and fraction using the decimal point as the break point. Then perform the appropriate conversion process on each number portion. After you convert the binary integer and binary fraction, recombine them. For example, the decimal number 14.375 is converted into its binary equivalent.

$$14.375_{10} = 14_{10} + 0.375_{10}$$

$$14 \div 2 = 7$$

$$7 \div 2 = 3$$

$$3 \div 2 = 1$$

$$1 \div 2 = 0$$

with remainder 0 ← LSB

1

1

1 ← MSB

$$\boxed{14_{10} = 1110_2}$$

$$0.375 \times 2 = 0.75 = 0.75$$

$$0.750 \times 2 = 1.50 = 0.50$$

$$0.500 \times 2 = 1.00 = 0$$

with overflow 0 ← MSB

1

1 ← LSB

$$\boxed{0.375_{10} = 0.011_2}$$

$$14.375_{10} = 14_{10} + 0.375_{10} = 1110_2 + 0.011_2 = 1110.011_2.$$

## Self-Test Review

7. The base or radix of the binary number system is \_\_\_\_\_.
8. A binary digit is called a \_\_\_\_\_.
9. Convert the following binary integers to decimal.
  - A. 101101
  - B. 1001
  - C. 1101100
10. Convert the following binary fractions to decimal.
  - A. 0.011
  - B. 0.01101
  - C. 0.1001
11. Convert the following decimal integers to binary.
  - A. 63
  - B. 12
  - C. 132
12. Convert the following decimal fractions to binary.
  - A. 0.4375
  - B. 0.96875
  - C. 0.625
13. Convert  $13.125_{10}$  to binary.

## Answers

7. 2

8. bit

9. A.  $101101_2 =$   
 $(1 \times 2^5) + (0 \times 2^4) + (1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) =$   
 $32 + 0 + 8 + 4 + 0 + 1 = 45_{10}$

B.  $1001_2 = 9_{10}$

C.  $1101100_2 = 108_{10}$

10. A.  $0.011_2 = (0 \times 2^{-1}) + (1 \times 2^{-2}) + (1 \times 2^{-3}) =$   
 $0 + \frac{1}{4} + \frac{1}{8} = 0 + 0.25 + 0.125 = 0.375_{10}$

B.  $0.01101_2 = 0.40625_{10}$

C.  $0.1001_2 = 0.5625_{10}$

11. A.  $63 \div 2 = 31$  with remainder 1 ← LSB  
 $31 \div 2 = 15$  1  
 $15 \div 2 = 7$  1  
 $7 \div 2 = 3$  1  
 $3 \div 2 = 1$  1  
 $1 \div 2 = 0$  1 ← MSB

$$\boxed{63_{10} = 111111_2}$$

B.  $12_{10} = 1100_2$

C.  $132_{10} = 10000100_2$



12. A.  $0.4375 \times 2 = 0.875 = 0.875$  with overflow 0  $\leftarrow$  MSB  
 $0.8750 \times 2 = 1.750 = 0.750$  1  
 $0.7500 \times 2 = 1.500 = 0.500$  1  
 $0.5000 \times 2 = 1.000 = 0$  1  $\leftarrow$  LSB

$$\boxed{0.4375_{10} = 0.0111_2}$$

B.  $0.96875_{10} = 0.11111_2$

C.  $0.625_{10} = 0.101_2$

13.  $13.125_{10} = 13_{10} + 0.125_{10}$

$13 \div 2 = 6$  with remainder 1  $\leftarrow$  LSB  
 $6 \div 2 = 3$  0  
 $3 \div 2 = 1$  1  
 $1 \div 2 = 0$  1  $\leftarrow$  MSB

$$\boxed{13_{10} = 1101_2}$$

$0.125 \times 2 = 0.25 = 0.25$  with overflow 0  $\leftarrow$  MSB  
 $0.250 \times 2 = 0.50 = 0.50$  0  
 $0.500 \times 2 = 1.00 = 0$  1  $\leftarrow$  LSB

$$\boxed{0.125_{10} = 0.001_2}$$

$$13.125_{10} = 13_{10} + 0.125_{10} = 1101_2 + 0.001_2 = 1101.001_2$$

## OCTAL NUMBER SYSTEM

Octal is another number system that is often used with microprocessors. It has a base (radix) of 8, and uses the digits 0 through 7. These eight digits have the same basic value as the digits 0—7 in the decimal number system.

As with the binary number system, each digit position of an octal number carries a positional weight which determines the magnitude of that number. The weight of each position is determined by some power of the number system base (in this example, 8). To evaluate the total quantity of a number, consider the specific digits and the weights of their positions. Refer to Figure 1-5 for a condensed listing of powers of 8. For example, the octal number 372.01 can be written with positional notation as follows:

$$(3 \times 8^2) + (7 \times 8^1) + (2 \times 8^0) + (0 \times 8^{-1}) + (1 \times 8^{-2})$$

The decimal value of the octal number 372.01 is determined by multiplying each digit by its positional weight and adding the results. As with decimal and binary numbers, the radix (octal) point separates the integer from the fractional part of the number.

$$(3 \times 64) + (7 \times 8) + (2 \times 1) + (0 \times 0.125) + (1 \times 0.015625) = 192 + 56 + 2 + 0 + 0.015625 = 250.015625_{10}$$

$$8^{-4} = \frac{1}{4096} = 0.000244140625_{10}$$

$$8^{-3} = \frac{1}{512} = 0.001953125_{10}$$

$$8^{-2} = \frac{1}{64} = 0.015625_{10}$$

$$8^{-1} = \frac{1}{8} = 0.125_{10}$$

$$1_{10} = 8^0$$

$$8_{10} = 8^1$$

$$64_{10} = 8^2$$

$$512_{10} = 8^3$$

$$4096_{10} = 8^4$$

$$32768_{10} = 8^5$$

$$262144_{10} = 8^6$$

Figure 1-5

Condensed listing of powers of 8.

## Conversion From Decimal to Octal

Decimal to octal conversion is accomplished in the same manner as decimal to binary, with one exception; the base number is now 8 rather than 2. As an example, the decimal number 194 is converted into its octal equivalent.

$$\begin{array}{rcll} 194 \div 8 = 24 \text{ with remainder } 2 & \leftarrow & \text{LSD} \\ 24 \div 8 = 3 & & 0 \\ 3 \div 8 = 0 & & 3 \leftarrow \text{MSD} \end{array}$$

Divide the decimal number by 8 and note the remainder. (The remainder can be any number from 0 to 7.)

Then divide the quotient by 8 and again note the remainder. Continue dividing until 0 results. Finally, collect the remainders beginning with the last or most significant digit (MSD) and proceed to the first or least significant digit (LSD). The number  $302_8 = 194_{10}$ . Figure 1-6 illustrates the relationship between the first several decimal, octal, and binary integers.

DECIMAL	OCTAL	BINARY
0	0	0
1	1	1
2	2	10
3	3	11
4	4	100
5	5	101
6	6	110
7	7	111
8	10	1000
9	11	1001
10	12	1010
11	13	1011
12	14	1100
13	15	1101
14	16	1110
15	17	1111
16	20	10000
17	21	10001
18	22	10010
19	23	10011
20	24	10100

Figure 1-6  
Sample comparison of decimal,  
octal, and binary integers.

To further illustrate this process, the decimal number 175 is converted into its octal equivalent.

$$\begin{array}{rcl}
 175 \div 8 = 21 \text{ with remainder } 7 & \leftarrow & \text{LSD} \\
 21 \div 8 = 2 & & 5 \\
 2 \div 8 = 0 & & 2 \leftarrow \text{MSD}
 \end{array}$$

The division process continues until a quotient of 0 results. The remainders are collected, producing the number  $257_8 = 175_{10}$ .

To convert a decimal fraction to an octal fraction, multiply the fraction successively by 8 (octal base). As an example, the decimal fraction 0.46875 is converted into its octal equivalent.

$$\begin{array}{rcl}
 0.46875 \times 8 = 3.75 = 0.75 \text{ with overflow } 3 & \leftarrow & \text{MSD} \\
 0.75000 \times 8 = 6.00 = 0 & & 6 \leftarrow \text{LSD}
 \end{array}$$

Multiply the decimal number by 8. If the product exceeds one, subtract the integer (overflow) from the product. Then multiply the product fraction by 8 and again note any "overflow." Continue multiplying until an overflow, with 0 for a fraction, results. Remember, you can not always obtain 0 when you multiply by 8. Therefore, you should only continue this conversion process to the accuracy or precision you desire. Collect the conversion overflows beginning at the radix (octal point) with the MSD and proceed to the LSD. The number  $0.36_8 = 0.46875_{10}$ . Figure 1-7 illustrates the relationship between decimal, octal, and binary fractions.

Now, the decimal fraction 0.136 will be converted into its octal equivalent with four-place precision.

$$\begin{array}{rcl}
 0.136 \times 8 = 1.088 = 0.088 \text{ with overflow } 1 & \leftarrow & \text{MSD} \\
 0.088 \times 8 = 0.704 = 0.704 & & 0 \\
 0.704 \times 8 = 5.632 = 0.632 & & 5 \\
 0.632 \times 8 = 5.056 = 0.056 & & 5 \leftarrow \text{LSD} \\
 0.136_{10} \approx 0.1055_8
 \end{array}$$

The number  $0.1055_8$  approximately equals  $0.136_{10}$ . If you convert  $0.1055_8$  back to decimal (using positional notation), you will find  $0.1055_8 = 0.135986328125_{10}$ . This example shows that extending the precision of your conversion is of little value unless extreme accuracy is required.

DECIMAL	OCTAL	BINARY
0.015625	0.01	0.000001
0.03125	0.02	0.00001
0.046875	0.03	0.000011
0.0625	0.04	0.0001
0.078125	0.05	0.000101
0.09375	0.06	0.00011
0.109375	0.07	0.000111
0.125	0.1	0.001
0.140625	0.11	0.001001
0.15625	0.12	0.00101
0.171875	0.13	0.001011
0.1875	0.14	0.0011
0.203125	0.15	0.001101
0.21875	0.16	0.00111
0.234375	0.17	0.001111
0.25	0.2	0.01
0.265625	0.21	0.010001
0.28125	0.22	0.01001
0.296875	0.23	0.010011
0.3125	0.24	0.0101

Figure 1-7  
 Sample comparison of decimal,  
 octal, and binary fractions.

As with decimal to binary conversion of a number that contains both an integer and fraction, decimal to octal conversion requires two operations. You must separate the integer from the fraction, then perform the appropriate conversion on each number. After you convert them, you must recombine the octal integer and octal fraction. For example, convert the decimal number 124.78125 into its octal equivalent.

$$124.78125_{10} = 124_{10} + 0.78125_{10}$$

$$124 \div 8 = 15 \quad \text{with remainder} \quad 4 \quad \leftarrow \text{LSD}$$

$$15 \div 8 = 1 \quad 7$$

$$1 \div 8 = 0 \quad 1 \quad \leftarrow \text{MSD}$$

$$124_{10} = 174_8$$

$$0.78125 \times 8 = 6.25 = 0.25 \quad \text{with overflow} \quad 6 \quad \leftarrow \text{MSD}$$

$$0.25000 \times 8 = 2.00 = 0 \quad 2 \quad \leftarrow \text{LSD}$$

$$0.78125_{10} = 0.62_8$$

$$124.78125_{10} = 124_{10} + 0.78125_{10} = 174_8 + 0.62_8 = 174.62_8$$

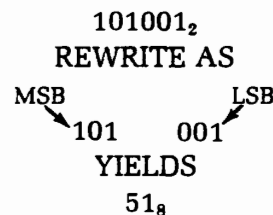
## Converting Between the Octal and Binary Number Systems

Microprocessors manipulate data using the binary number system. However, when larger quantities are involved, the binary number system can become cumbersome. Therefore, other number systems are frequently used as a form of binary shorthand to speed-up and simplify data entry and display. The octal number system is one of the systems that is used in this manner. It is similar to the decimal number system, which makes it easier to understand numerical values. In addition, conversion between binary and octal is readily accomplished because of the value structure of octal. Figures 1-6 and 1-7 illustrate the relationship between octal and binary integers and fractions.

As you know, three bits of a binary number exactly equal eight value combinations. Therefore, you can represent a 3-bit binary number with a 1-digit octal number.

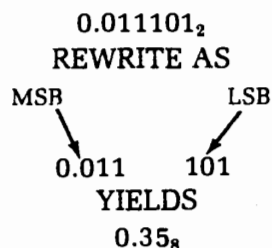
$$101_2 = (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) = 4 + 0 + 1 = 5_8$$

Because of this relationship, converting binary to octal is simple and straight forward. For example, binary number 101001 is converted into its octal equivalent.



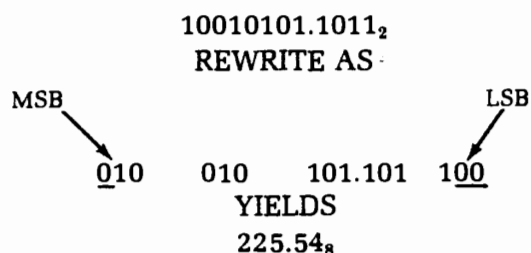
To convert a binary number to octal, first separate the number into groups containing three bits, beginning with the least significant bit. Then convert each 3-bit group into its octal equivalent. This gives you an octal number equal in value to the binary number.

Binary fractions can also be converted to their octal equivalents using the same process, with one exception. The binary bits must be separated into groups of three beginning with the most significant bit. For example, the binary fraction  $0.011101_2$  is converted into its octal equivalent.



Again, you must first separate the binary number into groups of three beginning at the radix (binary) point. Then convert each 3-bit group into its octal equivalent.

To separate binary numbers into 3-bit groups when the number does not contain the necessary bits, add zeros to the number until the number can be separated into 3-bit groups. For example, binary number  $10010101.1011_2$  is converted into its octal equivalent.



As before, the integer part of the number is separated into 3-bit groups, beginning at the radix (binary) point. Note that the third group contains only two bits. However, a zero can be added to the group without changing the value of the binary number. Next, the fractional part of the number is separated into 3-bit groups, beginning at the radix (binary) point. Note that the second group contains only one bit. By adding two zeros to the group, the group is complete with no change in the value of the binary number.

**NOTE:** Whenever you add zeros to a **binary integer**, always place them to the **left** of the most significant bit. When you add zeros to a **binary fraction**, always place them to the **right** of the least significant bit.

After you have formed the 3-bit groups, convert each group into its octal equivalent. This gives you an octal number equal in value to the binary number. Now convert binary number 1101110.01 into its octal equivalent.

1101110.01<sub>2</sub>  
REWRITE AS

MSB ↙	<u>001</u>	101	110.01	↘ LSB
----------	------------	-----	--------	----------

YIELDS  
156.2<sub>8</sub>

Separate the integer and fraction into 3-bit groups, adding zeros as necessary. Then convert each 3-bit group to octal. **Never** shift the radix (binary) point in order to form 3-bit groups.

Converting octal to binary is just the opposite of the previous process. You simply convert each octal number into its 3-bit binary equivalent. For example, convert the octal number 75.3 into its binary equivalent.

75.3<sub>8</sub>  
YIELDS

MSB ↙	111	101.011	↘ LSB
----------	-----	---------	----------

REWRITE AS  
111101.011<sub>2</sub>

The above example is a simple conversion. Now a more complex octal number (1752.714) will be converted to a binary number.

1752.714<sub>8</sub>  
YIELDS

MSB ↙	<u>001</u>	111	101	010.111	001	↘ LSB
----------	------------	-----	-----	---------	-----	----------

REWRITE AS  
1111101010.1110011<sub>2</sub>

Again, each octal digit is converted into its 3-bit binary equivalent. However, in this example, there are two insignificant zeros in front of the MSB and after the LSB. Since these zeros have no value, they should be removed from the final result.



## Self-Test Review

14. The base or radix of the octal number system is \_\_\_\_\_.
15. Convert the following decimal integers to octal.
  - A. 156
  - B. 32
  - C. 1785
16. Convert the following decimal fractions to octal. Do not use greater than 4-place precision.
  - A. 0.1432
  - B. 0.8125
  - C. 0.6832
17. Convert  $735.984375_{10}$  to octal.
18. Convert the following binary numbers to octal.
  - A. 10000111.01101
  - B. 11101.0101
  - C. 1001101.000001
19. Convert the following octal numbers to binary.
  - A. 372.61
  - B. 11.001
  - C. 3251.034

## Answers

14. 8

15. A.  $156 \div 8 = 19$  with remainder 4  $\leftarrow$  LSD  
 $19 \div 8 = 2$  3  
 $2 \div 8 = 0$  2  $\leftarrow$  MSD

$$156_{10} = 234_8$$

B.  $32_{10} = 40_8$ C.  $1785_{10} = 3371_8$ 

16. A.  $0.1432 \times 8 = 1.1456 = 0.1456$  overflow 1  $\leftarrow$  MSD  
 $0.1456 \times 8 = 1.1648 = 0.1648$  1  
 $0.1648 \times 8 = 1.3184 = 0.3184$  1  
 $0.3184 \times 8 = 2.5472 = 0.5472$  2  $\leftarrow$  LSD

$$0.1432_{10} = 0.1112_8$$

B.  $0.8125_{10} = 0.64_8$ C.  $0.6832_{10} = 0.5356_8$ 

17.  $735.984375_{10} = 735_{10} + 0.984375_{10}$   
 $= 735 \div 8 = 91$  with remainder 7  $\leftarrow$  LSD  
 $91 \div 8 = 11$  3  
 $11 \div 8 = 1$  3  
 $1 \div 8 = 0$  1  $\leftarrow$  MSD

$$735_{10} = 1337_8$$

$0.984375 \times 8 = 7.875 = 0.875$  overflow  
 7  $\leftarrow$  MSD  
 $0.875000 \times 8 = 7.00 = 0$  7  $\leftarrow$  LSD

$$0.984375_{10} = 0.77_8$$

$$735.984375_{10} = 735_{10} + 0.984375_{10} = 1337_8 + 0.77_8 = 1337.77_8$$

18. A.  $10000111.01101_2 = 010\ 000\ 111.011\ 010_2$   
 $= 207.32_8$

B.  $11101.0101_2 = 35.24_8$ C.  $1001101.000001_2 = 115.01_8$ 

19. A.  $372.61_8 = 011\ 111\ 010.110\ 001_2 = 11111010.110001_2$   
 B.  $11.001_8 = 1001.000000001_2$   
 C.  $3251.034_8 = 11010101001.0000111_2$

## HEXADECIMAL NUMBER SYSTEM

Hexadecimal is another number system that is often used with microprocessors. It is similar in value structure to the octal number system, and thus allows easy conversion with the binary number system. Because of this feature and the fact that hexadecimal simplifies data entry and display to a greater degree than octal, you will use hexadecimal more often than any other number system in this course. As the name implies, hexadecimal has a base (radix) of  $16_{10}$ . It uses the digits 0 through 9 and the letters A through F.

The letters are used because it is necessary to represent  $16_{10}$  different values with a single digit for each value. Therefore, the letters A through F are used to represent the number values  $10_{10}$  through  $15_{10}$ . The following discussion will compare the decimal number system with the hexadecimal number system.

All of the numbers are of equal value between systems ( $0_{10} = 0_{16}$ ,  $3_{10} = 3_{16}$ ,  $9_{10} = 9_{16}$ , etc.). For numbers greater than 9, this relationship exists:  $10_{10} = A_{16}$ ,  $11_{10} = B_{16}$ ,  $12_{10} = C_{16}$ ,  $13_{10} = D_{16}$ ,  $14_{10} = E_{16}$ , and  $15_{10} = F_{16}$ . Using letters in counting may appear awkward until you become familiar with the system. Figure 1-8 illustrates the relationship between decimal and hexadecimal integers, while Figure 1-9 illustrates the relationship between decimal and hexadecimal fractions.

DECIMAL	HEXADECIMAL	BINARY
0	0	0
1	1	1
2	2	10
3	3	11
4	4	100
5	5	101
6	6	110
7	7	111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111
16	10	10000
17	11	10001
18	12	10010
19	13	10011
20	14	10100
21	15	10101
22	16	10110
23	17	10111
24	18	11000
25	19	11001
26	1A	11010
27	1B	11011
28	1C	11100
29	1D	11101
30	1E	11110
31	1F	11111
32	20	100000
33	21	100001
34	22	100010
35	23	100011

Figure 1-8  
Sample comparison of decimal,  
hexadecimal, and binary integers.

DECIMAL	HEXADECIMAL	BINARY
0.00390625	0.01	0.00000001
0.0078125	0.02	0.0000001
0.01171875	0.03	0.00000011
0.015625	0.04	0.000001
0.01953125	0.05	0.00000101
0.0234375	0.06	0.0000011
0.02734375	0.07	0.00000111
0.03125	0.08	0.00001
0.03515625	0.09	0.00001001
0.0390625	0.0A	0.0000101
0.04296875	0.0B	0.00001011
0.046875	0.0C	0.00011
0.05078125	0.0D	0.00001101
0.0546875	0.0E	0.0000111
0.05859375	0.0F	0.00001111
0.0625	0.1	0.0001
0.06640625	0.11	0.00010001
0.0703125	0.12	0.0001001
0.07421875	0.13	0.00010011
0.078125	0.14	0.000101
0.08203125	0.15	0.00010101
0.0859375	0.16	0.0001011
0.08984375	0.17	0.00010111
0.09375	0.18	0.00011
0.09765625	0.19	0.00011001
0.1015625	0.1A	0.0001101
0.10546875	0.1B	0.00011011
0.109375	0.1C	0.000111
0.11328125	0.1D	0.00011101
0.1171875	0.1E	0.0001111
0.12109375	0.1F	0.00011111
0.125	0.2	0.001

Figure 1-9

Sample comparison of decimal,  
hexadecimal, and binary fractions.

As with the previous number systems, each digit position of a hexadecimal number carries a positional weight which determines the magnitude of that number. The weight of each position is determined by some power of the number system base (in this example,  $16_{10}$ ). The total quantity of a number can be evaluated by considering the specific digits and the weights of their positions. (Refer to Figure 1-10 for a condensed listing of powers of  $16_{10}$ .) For example, the hexadecimal number E5D7.A3 can be written with positional notation as follows:

$$(E \times 16^3) + (5 \times 16^2) + (D \times 16^1) + (7 \times 16^0) + (A \times 16^{-1}) + (3 \times 16^{-2})$$

The decimal value of the hexadecimal number E5D7.A3 is determined by multiplying each digit by its positional weight and adding the results. As with the previous number systems, the radix (hexadecimal) point separates the integer from the fractional part of the number.

$$\begin{aligned} &(14 \times 4096) + (5 \times 256) + (13 \times 16) + (7 \times 1) + (10 \times 1/16) + (3 \times 1/256) = \\ &57344 + 1280 + 208 + 7 + 0.625 + 0.01171875 = \\ &58839.63671875_{10} \end{aligned}$$

$$\begin{aligned} 16^{-4} &= \frac{1}{65536} = 0.0000152587890625_{10} \\ 16^{-3} &= \frac{1}{4096} = 0.000244140625_{10} \\ 16^{-2} &= \frac{1}{256} = 0.00390625_{10} \\ 16^{-1} &= \frac{1}{16} = 0.0625_{10} \end{aligned}$$

$$\begin{aligned} 1_{10} &= 16^0 \\ 16_{10} &= 16^1 \\ 256_{10} &= 16^2 \\ 4096_{10} &= 16^3 \\ 65536_{10} &= 16^4 \\ 1048576_{10} &= 16^5 \\ 16777216_{10} &= 16^6 \end{aligned}$$

Figure 1-10  
Condensed listing of powers of 16.

## Conversion From Decimal to Hexadecimal

Decimal to hexadecimal conversion is accomplished in the same manner as decimal to binary or octal, but with a base number of  $16_{10}$ . As an example, the decimal number 156 is converted into its hexadecimal equivalent.

$$\begin{array}{rcll} 156 \div 16 = 9 & \text{with remainder } 12 = C & \leftarrow & \text{LSD} \\ 9 \div 16 = 0 & & 9 = 9 & \leftarrow \text{MSD} \end{array}$$

Divide the decimal number by  $16_{10}$  and note the remainder. If the remainder exceeds 9, convert the 2-digit number to its hexadecimal equivalent ( $12_{10} = C$  in this example). Then divide the quotient by 16 and again note the remainder. Continue dividing until a quotient of 0 results. Then collect the remainders beginning with the last or most significant digit (MSD) and proceed to the first or least significant digit (LSD). The number  $9C_{16} = 156_{10}$ . NOTE: The letter H after a number is sometimes used to indicate hexadecimal. However, this course will always use the subscript 16.

To further illustrate this, the decimal number 47632 is converted into its hexadecimal equivalent.

$$\begin{array}{rcll} 47632 \div 16 = 2977 & \text{with remainder } 0 = 0 & \leftarrow & \text{LSD} \\ 2977 \div 16 = 186 & & 1 = 1 & \\ 186 \div 16 = 11 & & 10 = A & \\ 11 \div 16 = 0 & & 11 = B & \leftarrow \text{MSD} \end{array}$$

The division process continues until a quotient of 0 results. The remainders are collected, producing the number  $BA10_{16} = 47632_{10}$ . Remember, any remainder that exceeds the digit 9 must be converted to its letter equivalent. (In this example,  $10 = A$ , and  $11 = B$ .)

To convert a decimal fraction to a hexadecimal fraction, multiply the fraction successively by  $16_{10}$  (hexadecimal base). As an example the decimal fraction 0.78125 is converted into its hexadecimal equivalent.

$$\begin{array}{rcll} 0.78125 \times 16 = 12.5 = 0.5 & \text{with overflow } 12 = C & \leftarrow & \text{MSD} \\ 0.50000 \times 16 = 8.0 = 0 & & 8 = 8 & \leftarrow \text{LSD} \end{array}$$

Multiply the decimal by  $16_{10}$ . If the product exceeds one, subtract the integer (overflow) from the product. If the "overflow" exceeds 9, convert the 2-digit number to its hexadecimal equivalent. Then multiply the product fraction by  $16_{10}$  and again note any overflow. Continue multiplying until an overflow, with 0 for a fraction, results. Remember, you can not always obtain 0 when you multiply by 16. Therefore, you should only continue the conversion to the accuracy or precision you desire. Collect the conversion overflows beginning at the radix point with the MSD and proceed to the LSD. The number  $0.C8_{16} = 0.78125_{10}$ .

Now the decimal fraction 0.136 will be converted into its hexadecimal equivalent with five-place precision.

$0.136 \times 16 = 2.176 = 0.176$	overflow	$2 = 2 \rightarrow$	MSD
$0.176 \times 16 = 2.816 = 0.816$		$2 = 2$	
$0.816 \times 16 = 13.056 = 0.056$		$13 = D$	
$0.056 \times 16 = 0.896 = 0.896$		$0 = 0$	
$0.896 \times 16 = 14.336 = 0.336$		$14 = E \rightarrow$	LSD

The number  $0.22D0E_{16}$  approximately equals  $0.136_{10}$ . If you convert  $0.22D0E_{16}$  back to decimal (using positional notation), you will find  $0.22D0E_{16} = 0.1359996795654296875_{16}$ . This example shows that extending the precision of your conversion is of little value unless extreme accuracy is required.



As shown in this section, conversion of an integer from decimal to hexadecimal requires a different technique than for conversion of a fraction. Therefore, when you convert a hexadecimal number composed of an integer and a fraction, you must separate the integer and fraction, then perform the appropriate operation on each. After you convert them, you must recombine the integer and fraction. For example, the decimal number 124.78125 is converted into its hexadecimal equivalent.

$$124.78125_{10} = 124_{10} + 0.78125_{10}$$

$$124 \div 16 = 7 \quad \text{with remainder } 12 = C \quad \leftarrow \text{LSD}$$

$$7 \div 16 = 0 \quad \quad \quad 7 = 7 \quad \leftarrow \text{MSD}$$

$$124_{10} = 7C_{16}$$

$$0.78125 \times 16 = 12.5 = 0.5 \quad \text{overflow} \quad 12 = C \quad \leftarrow \text{MSD}$$

$$0.50000 \times 16 = 8.0 = 0 \quad \quad \quad 8 = 8 \quad \leftarrow \text{LSD}$$

$$0.78125_{10} = 0.C8_{16}$$

$$124.78125_{10} = 124_{10} + 0.78125_{10} = 7C_{16} + 0.C8_{16} = 7C.C8_{16}$$

First separate the decimal integer and fraction. Then convert the integer and fraction to hexadecimal.

Finally, recombine the integer and fraction.

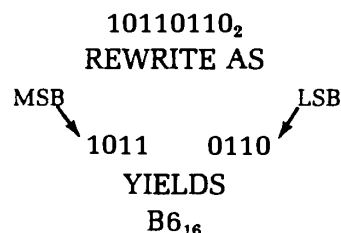
## Converting Between the Hexadecimal and Binary Number Systems

Previously, the octal number system was described as an excellent shorthand form to express large binary quantities. This method is very useful with many microprocessors. The trainer used with this course uses the hexadecimal number system to represent binary quantities. As a result, frequent conversions from binary-to-hexadecimal are necessary. Figures 1-8 and 1-9 illustrate the relationship between hexadecimal and binary integers and fractions.

As you know, four bits of a binary number exactly equal  $16_{10}$  value combinations. Therefore, you can represent a 4-bit binary number with a 1-digit hexadecimal number:

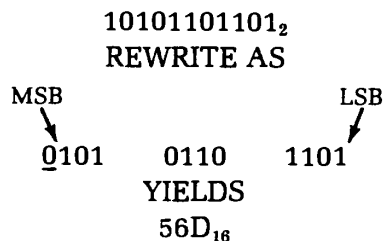
$$1101_2 = (1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) = 8 + 4 + 0 + 1 = 13_{10} = D_{16}$$

Because of this relationship, converting binary to hexadecimal is simple and straightforward. For example, binary number 10110110 is converted into its hexadecimal equivalent.



To convert a binary number to hexadecimal, first separate the number into groups containing four bits, beginning with the least significant bit. Then convert each 4-bit group into its hexadecimal equivalent. Don't forget to use letter digits as required. This gives you a hexadecimal number equal in value to the binary number.

Now convert a larger binary number (10101101101) into its hexadecimal equivalent.



Again, the binary number is separated into 4-bit groups beginning with the LSB. However, the third group contains only three bits. Since each group must contain four bits, a zero must be added after the MSB. The third group will then have four bits with no change in the value of the binary number. Now each 4-bit group can be converted into its hexadecimal equivalent. **Whenever you add zeros to a binary integer, always place them to the left of the most significant bit.**

Binary fractions can also be converted to their hexadecimal equivalents using the same process, with one exception; the binary bits are separated into groups of four, beginning with the most significant bit (at the radix point). For example, the binary fraction 0.01011011 is converted into its hexadecimal equivalent.

$$\begin{array}{c} 0.01011011_2 \\ \text{REWRITE AS} \\ \begin{array}{cc} \text{MSB} \swarrow & \searrow \text{LSB} \\ 0.0101 & 1011 \end{array} \\ \text{YIELDS} \\ 0.5B_{16} \end{array}$$

Again, you must separate the binary number into groups of four, beginning with the radix point. Then convert each 4-bit group into its hexadecimal equivalent. This gives you a hexadecimal number equal in value to the binary number.

Now convert a larger binary fraction (0.1101001101) into its hexadecimal equivalent.

$$\begin{array}{c} 0.1101001101_2 \\ \text{REWRITE AS} \\ \begin{array}{ccc} \text{MSB} \swarrow & & \searrow \text{LSB} \\ 0.1101 & 0011 & 0100 \end{array} \\ \text{YIELDS} \\ 0.D34_{16} \end{array}$$

Separate the binary number into 4-bit groups, beginning at the radix (binary) point (MSB). Note that the third group contains only two bits. Since each group must contain four bits, two zeros must be added after the LSB. The third group will then have four bits with no change in the value of the binary number. Now, each 4-bit group can be converted into its hexadecimal equivalent. **Whenever you add zeros to a binary fraction, always place them to the right of the least significant bit.**

Now, a binary number containing both an integer and a fraction ( $110110101.01110111_2$ ) will be converted into its hexadecimal equivalent.

$110110101.01110111_2$   
REWRITE AS

MSB ↓	<u>0001</u>	1011	0101.0111	0111 ↑ LSB
----------	-------------	------	-----------	------------------

YIELDS  
 $1B5.77_{16}$

The integer part of the number is separated into groups of four, **beginning** at the radix point. Note that three zeros were added to the third group to complete the group. The fractional part of the number is separated into groups of four, **beginning** at the radix point. (No zeros were needed to complete the fractional groups.) The integer and fractional 4-bit groups are then converted to hexadecimal. The number  $110110101.01110111_2 = 1B5.77_{16}$ . **Never** shift the radix point in order to form 4-bit groups.

Converting hexadecimal to binary is just the opposite of the previous process; simply convert each hexadecimal number into its 4-bit binary equivalent. For example, convert the hexadecimal number  $8F.41_{16}$  into its binary equivalent.

$8F.41_{16}$   
YIELDS

MSB ↓	1000	1111.0100	0001 ↑ LSB
----------	------	-----------	------------------

REWRITE AS  
 $10001111.01000001_2$

Convert each hexadecimal digit into a 4-bit binary number. Then condense the 4-bit groups to form the binary value equal to the hexadecimal value. The number  $8F.41_{16} = 10001111.01000001_2$ .

Now, the hexadecimal number 175.4E will be converted into its binary equivalent.

175.4E<sub>16</sub>  
YIELDS  
MSB                      LSB  
↓                              ↓  
0001 0111 0101.0100 1110  
REWRITE AS  
101110101.0100111<sub>2</sub>

Again, each hexadecimal digit is converted into its 4-bit binary equivalent. However, in this example there are three insignificant zeros in front of the MSB and one after the LSB. Since these zeros have no value, they should be removed from the final result.

## Self-Test Review

20. The base or radix of the hexadecimal number system is \_\_\_\_\_.
21. Convert the following decimal integers to hexadecimal.
- A. 783
  - B. 5372
  - C. 957
22. Convert the following decimal fractions to hexadecimal. Do not use greater than four-place precision.
- A. 0.653
  - B. 0.109375
  - C. 0.4567
23. Convert  $1573.125_{10}$  to its hexadecimal equivalent.
24. Convert the following binary numbers to hexadecimal.
- A. 100001101.01011
  - B. 11111011001.01
  - C. 110001101.00010010101
25. Convert the following hexadecimal numbers to binary.
- A. AE7.D2
  - B. 2C5.21F8
  - C. 1B6.64E

## Answers

20.  $16_{10}$

21. A.  $783 \div 16 = 48$  with remainder  $15 = F \leftarrow \text{LSD}$   
 $48 \div 16 = 3 \quad 0 = 0$   
 $3 \div 16 = 0 \quad 3 = 3 \leftarrow \text{MSD}$

$$783_{10} = 30F_{16}$$

B.  $5372_{10} = 14FC_{16}$

C.  $957_{10} = 3BD_{16}$

22. A.  $0.653 \times 16 = 10.448 = 0.448$  with overflow  $10 = A \leftarrow \text{MSD}$   
 $0.448 \times 16 = 7.168 = 0.168 \quad 7 = 7$   
 $0.168 \times 16 = 2.688 = 0.688 \quad 2 = 2$   
 $0.688 \times 16 = 11.008 = 0.008 \quad 11 = B \leftarrow \text{LSD}$

$$0.653_{10} = 0.A72B_{16}$$

B.  $0.109375_{10} = 0.1C_{16}$

C.  $0.4567_{10} = 0.74EA_{16}$

23. A.  $1573.125_{10} = 1573_{10} + 0.125_{10}$   
 $1573 \div 16 = 98$  with remainder  $5 = 5 \leftarrow \text{LSD}$   
 $98 \div 16 = 6 \quad 2 = 2$   
 $6 \div 16 = 0 \quad 6 = 6 \leftarrow \text{MSD}$

$$1573_{10} = 625_{16}$$

$0.125 \times 16 = 2.00 = 0$  with overflow  $2 = 2 \leftarrow \begin{matrix} \text{MSD} \\ \text{LSD} \end{matrix}$

$$0.125_{10} = 0.2_{16}$$

$$1573.125_{10} = 1573_{10} + 0.125_{10} = 625_{16} + 0.2_{16} = 625.2_{16}$$

24. A.  $100001101.01011_2 = 0001\ 0000\ 1101.0101\ 1000_2$   
 $= 10D.58_{16}$

B.  $11111011001.01_2 = 7D9.4_{16}$

C.  $110001101.00010010101_2 = 18D.12A_{16}$

25. A.  $AE7.D2_{16} = 1010\ 1110\ 0111.1101\ 0010_2$   
 $= 101011100111.1101001_2$   
 B.  $2C5.21F8_{16} = 1011000101.0010000111111_2$   
 C.  $1B6.64E_{16} = 110110110.01100100111_2$

## BINARY CODES

Converting a decimal number into its binary equivalent is called "coding." A decimal number is expressed as a binary code or binary number. The **binary number system**, as discussed, is known as the pure binary code. This name distinguishes it from other types of binary codes. This section will discuss some of the other types of binary codes used in computers.

### Binary Coded Decimal

The decimal number system is easy to use because it is so familiar. The binary number system is less convenient to use because it is less familiar. It is difficult to quickly glance at a binary number and recognize its decimal equivalent. For example, the binary number 1010011 represents the decimal number 83. It is difficult to tell immediately by looking at the number what its decimal value is. However, within a few minutes, using the procedures described earlier, you could readily calculate its decimal value. The amount of time it takes to convert or recognize a binary number quantity is a distinct disadvantage in working with this code despite the numerous hardware advantages. Engineers recognized this problem early and developed a special form of binary code that was more compatible with the decimal system. Because so many digital devices, instruments and equipment use decimal input and output, this special code has become very widely used and accepted. This special compromise code is known as binary coded decimal (BCD). The BCD code combines some of the characteristics of both the binary and decimal number systems.

**8421 BCD Code** The BCD code is a system of representing the decimal digits 0 through 9 with a four-bit binary code. This BCD code uses the standard 8421 position **weighting system** of the pure binary code. The standard 8421 BCD code and the decimal equivalents are shown in Figure 1-11, along with a special Gray code that will be described later. As with the pure binary code, you can convert the BCD numbers into their decimal equivalents by simply adding together the weights of the bit positions whereby the binary 1's occur. Note, however, that there are only ten possible valid 4-bit code arrangements. The 4-bit binary numbers representing the decimal numbers 10 through 15 are invalid in the BCD system.



DECIMAL	8421 BCD	GRAY	BINARY
0	0000	0000	0000
1	0001	0001	0001
2	0010	0011	0010
3	0011	0010	0011
4	0100	0110	0100
5	0101	0111	0101
6	0110	0101	0110
7	0111	0100	0111
8	1000	1100	1000
9	1001	1101	1001
10	0001 0000	1111	1010
11	0001 0001	1110	1011
12	0001 0010	1010	1100
13	0001 0011	1011	1101
14	0001 0100	1001	1110
15	0001 0101	1000	1111

Figure 1-11  
Codes.

To represent a decimal number in BCD notation, substitute the appropriate 4-bit code for each decimal digit. For example, the decimal integer 834 in BCD would be 1000 0011 0100. Each decimal digit is represented by its equivalent 8421 4-bit code. A space is left between each 4-bit group to avoid confusing the BCD format with the pure binary code. This method of representation also applies to decimal fractions. For example, the decimal fraction 0.764 would be 0.0111 0110 0100 in BCD. Again, each decimal digit is represented by its equivalent 8421 4-bit code, with a space between each group.

An advantage of the BCD code is that the ten BCD code combinations are easy to remember. Once you begin to work with binary numbers regularly, the BCD numbers may come to you as quickly and automatically as decimal numbers. For that reason, by simply glancing at the BCD representation of a decimal number you can make the conversion almost as quickly as if it were already in decimal form. As an example, convert a BCD number into its decimal equivalent.

$$0110\ 0010\ 1000.1001\ 0101\ 0100 = 628.954_{10}$$

The BCD code simplifies the man-machine interface but it is less efficient than the pure binary code. It takes more bits to represent a given decimal number in BCD than it does with pure binary notation. For example, the decimal number 83 in pure binary form is 1010011. In BCD code the decimal number 83 is written as 1000 0011. In the pure binary code, it takes only seven bits to represent the number 83. In BCD form, it takes eight bits. It is inefficient because, for each bit in a data word, there is usually some digital circuitry associated with it. The extra circuitry associated with the BCD code costs more, increases equipment complexity, and consumes more power. Arithmetic operations with BCD numbers are also more time consuming and complex than those with pure binary numbers. With four bits of binary information, you can represent a total of  $2^4 = 16$  different states or the decimal number equivalents 0 through 15. In the BCD system, six of these states (10-15), are wasted. When the BCD number system is used, some efficiency is traded for the improved communications between the digital equipment and the human operator.

Decimal-to-BCD conversion is simple and straightforward. However, binary-to-BCD conversion is not direct. An intermediate conversion to decimal must be performed first. For example, the binary number 1011.01 is converted into its BCD equivalent.

First the binary number is converted to decimal.

$$\begin{aligned} 1011.01_2 &= (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0) + (0 \times 2^{-1}) + (1 \times 2^{-2}) \\ &= 8 + 0 + 2 + 1 + 0 + 0.25 \\ &= 11.25_{10} \end{aligned}$$

Then the decimal result is converted to BCD.

$$11.25_{10} = 0001\ 0001.0010\ 0101$$

To convert from BCD to binary, the previous operation is reversed. For example, the BCD number 1001 0110.0110 0010 0101 is converted into its binary equivalent.

First, the BCD number is converted to decimal.

$$1001\ 0110.0110\ 0010\ 0101 = 96.625_{10}$$

Then the decimal result is converted to binary.

$$96.625_{10} = 96_{10} + 0.625_{10}$$

$96 \div 2 = 48$	with remainder	0	← LSB
$48 \div 2 = 24$		0	
$24 \div 2 = 12$		0	
$12 \div 2 = 6$		0	
$6 \div 2 = 3$		0	
$3 \div 2 = 1$		1	
$1 \div 2 = 0$		1	← MSB

$$96_{10} = 1100000_2$$

$0.625 \times 2 = 1.25$	$= 0.25$	with overflow	1	← MSB
$0.250 \times 2 = 0.50$	$= 0.50$		0	
$0.500 \times 2 = 1.00$	$= 0$		1	← LSB

$$0.625_{10} = 0.101_2$$

$$96.625_{10} = 96_{10} + 0.625_{10} = 1100000_2 + 0.101_2 = 1100000.101_2$$

Therefore:

$$1001\ 0110.0110\ 0010\ 0101 = 96.625_{10} = 1100000.101_2$$

Because the intermediate decimal number contains both an integer and fraction, each number portion is converted as described under "Binary Number System." The binary sum (integer plus fraction) 1100000.101 is equivalent to the BCD number 1001 0110.0110 0010 0101.

DECIMAL	8421 BCD	GRAY	BINARY
0	0000	0000	0000
1	0001	0001	0001
2	0010	0011	0010
3	0011	0010	0011
4	0100	0110	0100
5	0101	0111	0101
6	0110	0101	0110
7	0111	0100	0111
8	1000	1100	1000
9	1001	1101	1001
10	0001 0000	1111	1010
11	0001 0001	1110	1011
12	0001 0010	1010	1100
13	0001 0011	1011	1101
14	0001 0100	1001	1110
15	0001 0101	1000	1111

Figure 1-11  
Codes.

## Special Binary Codes

Besides the standard pure binary coded form, the BCD numbering system is by far the most widely-used digital code. You will find one or the other in most of the applications that you encounter. However, there are several other codes that are used for special applications, such as the "Gray Code."

The Gray Code is a widely-used, non-weighted code system. Also known as the cyclic, unit distance or reflective code, the Gray code can exist in either the pure binary or BCD formats. The Gray code is shown in Figure 1-11. As with the pure binary code, the first ten codes are used in BCD operations. Notice that there is a change in only one bit from one code number to the next in sequence. You can get a better idea about the Gray code sequence by comparing it to the standard 4-bit 8421 BCD code and the pure binary code also shown in Figure 1-11. For example, consider the change from 7 (0111) to 8 (1000) in the pure binary code. When this change takes place, all bits change. Bits that were 1's are changed to 0's and 0's are changed to 1's. Now notice the code change from 7 to 8 in the Gray code. Here 7 (0100) changes to 8 (1100). Only the first bit changes.

The Gray code is generally known as an error minimizing code because it greatly reduces confusion in the electronic circuitry when changing from one state to the next. When binary codes are implemented with electronic circuitry, it takes a finite period of time for bits to change from 0 to 1 or 1 to 0. These state changes can create timing and speed problems. This is particularly true in the standard 8421 codes where many bits change from one combination to the next. When the Gray code is used, however, the timing and speed errors are greatly minimized because only one bit changes at a time. This permits code circuitry to operate at higher speeds with fewer errors.

The biggest disadvantage of the Gray code is that it is difficult to use in arithmetic computations. Where numbers must be added, subtracted or used in other computations, the Gray code is not applicable. In order to perform arithmetic operations, the Gray code number must generally be converted into pure binary form.

## Alphanumeric Codes

Several binary codes are called alphanumeric codes because they are used to represent characters as well as numbers. The two most common codes that will be discussed are ASCII and BAUDOT.

**ASCII Code** The American Standard Code for Information Interchange commonly referred to as ASCII, is a special form of binary code that is widely used in microprocessors and data communications equipment. A new name for this code that is becoming more popular is the American National Standard Code for Information Interchange (ANSI). However, this course will use the most recognized term, ASCII. ASCII is a 6-bit binary code that is used in transferring data between microprocessors and their peripheral devices, and in communicating data by radio and telephone. With six bits, a total of  $2^6 = 64$  different characters can be represented. These characters comprise decimal numbers 0 through 9, upper-case letters of the alphabet, plus other special characters used for punctuation and data control. A 7-bit code called full ASCII, extended ASCII, or USASCII can be represented by  $2^7 = 128$  different characters. In addition to the characters and numbers generated by 6-bit ASCII, 7-bit ASCII contains lower-case letters of the alphabet, and additional characters for punctuation and control. The 7-bit ASCII code is shown in Figure 1-12.

COLUMN		0 <sup>(3)</sup>	1 <sup>(3)</sup>	2 <sup>(3)</sup>	3	4	5	6	7 <sup>(3)</sup>
ROW	BITS 4321 765	000	001	010	011	100	101	110	111
0	0000	NUL	DLE	SP	0	@	P	\	p
1	0001	SOH	DC1	!	1	A	Q	a	q
2	0010	STX	DC2	"	2	B	R	b	r
3	0011	ETX	DC3	#	3	C	S	c	s
4	0100	EOT	DC4	\$	4	D	T	d	t
5	0101	ENQ	NAK	%	5	E	U	e	u
6	0110	ACK	SYN	&	6	F	V	f	v
7	0111	BEL	ETB	'	7	G	W	g	w
8	1000	BS	CAN	(	8	H	X	h	x
9	1001	HT	EM	)	9	I	Y	i	y
10	1010	LF	SUB	*	:	J	Z	j	z
11	1011	VT	ESC	+	;	K	[	k	{
12	1100	FF	FS	,	<	L	\	l	!
13	1101	CR	GS	-	=	M		m	}
14	1110	SO	RS	.	>	N	~ <sup>(1)</sup>	n	~
15	1111	SI	US	/	?	O	— <sup>(2)</sup>	o	DEL

Figure 1-12

Table of 7-bit American Standard Code  
for Information Interchange.

## NOTES:

- (1) Depending on the machine using this code, the symbol may be a circumflex, an up-arrow, or a horizontal parenthetical mark.
- (2) Depending on the machine using this code, the symbol may be an underline, a back-arrow, or a heart.
- (3) Explanation of special control functions in columns 0, 1, 2, and 7.

NUL	Null	DLE	Data Link Escape
SOH	Start of Heading	DC1	Device Control 1
STX	Start of Text	DC2	Device Control 2
ETX	End of Text	DC3	Device Control 3
EOT	End of Transmission	DC4	Device Control 4
ENQ	Enquiry	NAK	Negative Acknowledge
ACK	Acknowledge	SYN	Synchronous Idle
BEL	Bell (audible signal)	ETB	End of Transmission Block
BS	Backspace	CAN	Cancel
HT	Horizontal Tabulation (punched card skip)	EM	End of Medium
LF	Line Feed	SUB	Substitute
VT	Vertical Tabulation	ESC	Escape
FF	Form Feed	FS	File Separator
CR	Carriage Return	GS	Group Separator
SO	Shift Out	RS	Record Separator
SI	Shift In	US	Unit Separator
SP	Space (blank)	DEL	Delete

Figure 1-12  
(Continued.)

The 7-bit ASCII code for each number, letter or control function is made up of a 4-bit group and a 3-bit group. Figure 1-13 shows the arrangement of these two groups and the numbering sequence. The 4-bit group is on the right and bit 1 is the LSB. Note how these groups are arranged in rows and columns in Figure 1-12.

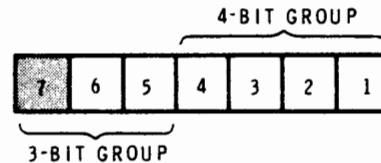


Figure 1-13  
ASCII code word format.

To determine the ASCII code for a given number letter or control operation, locate that item in the table. Then use the 3- and 4-bit codes associated with the row and column in which the item is located. For example, the ASCII code for the letter L is 1001100. It is located in column 4, row 12. The most significant 3-bit group is 100, while the least significant 4-bit group is 1100. When 6-bit ASCII is used, the 3-bit group is reduced to a 2-bit group as shown in Figure 1-14.

In 7-bit ASCII code, an eighth bit is often used as a **parity** or check bit to determine if the data (character) has been transmitted correctly. The value of this bit is determined by the type of parity desired. **Even parity** means the sum of all the 1 bits, including the parity bit, is an even number. For example, if G is the character transmitted, the ASCII code is 1000111. Since four 1's are in the code, the parity bit is 0. The 8-bit code would be written 01000111.

**OddParity** means the sum of all the 1 bits, including the parity bit, is an odd number. If the ASCII code for G was transmitted with odd parity, the binary representation would be 11000111.



COLUMN					
0 1 2 3					
ROW	BITS 4321 65	10	11	00	01
0	0000	SP <sup>(3)</sup>	0	@	P
1	0001	!	1	A	Q
2	0010	"	2	B	R
3	0011	#	3	C	S
4	0100	\$	4	D	T
5	0101	%	5	E	U
6	0110	&	6	F	V
7	0111	'	7	G	W
8	1000	(	8	H	X
9	1001	)	9	I	Y
10	1010	*	:	J	Z
11	1011	+	;	K	
12	1100	,	<	L	\
13	1101	-	=	M	
14	1110	.	>	N	⌒ <sup>(1)</sup>
15	1111	/	?	O	— <sup>(2)</sup>

**Figure 1-14**  
 Table of 6-bit American Standard Code  
 for Information Interchange.

**NOTES:**

- (1) Depending on the machine using this code, the symbol may be a circumflex, an up-arrow, or a horizontal parenthetical mark.
- (2) Depending on the machine using this code, the symbol may be an underline, a back-arrow, or a heart.
- (3) SP — Space (blank) for machine control.

**BAUDOT Code** While the ASCII code is used almost exclusively with microprocessor peripheral devices (CRT display, keyboard terminal, paper punch/reader, etc.), there are many older printer peripherals that use the 5-bit BAUDOT code. With five data bits, this code can represent only  $2^5 = 32$  different characters. To obtain a greater character capability, 26 of the 5-bit codes are used to represent two separate characters. As shown in Figure 1-15, one set of 5-bit codes represents the 26 upper-case alphabet letters. The same 5-bit codes also represent various figures and the decimal number series 0 through 9.

The remaining six 5-bit codes are used for machine control and do not have a secondary function. Two of these 5-bit codes determine which of the 26 double (letter/figure) characters can be transmitted/received. Bit number 11111 forces the printer to recognize all following 5-bit codes as **letters**. Bit number 11011 forces **figure** recognition of all the following 5-bit codes. For example, to type 56 NORTH 10 STREET, the following method is used.

Type — Figures 5 6 Space

Then — Letters N O R T H Space

Then — Figures 1 0 Space

Finally — Letters S T R E E T

Bit Numbers 5 4 3 2 1	Letters Case	Figures Case
0 0 0 0 0	Blank	Blank
0 0 0 0 1	E	3
0 0 0 1 0	Line Feed	Line Feed
0 0 0 1 1	A	—
0 0 1 0 0	Space	Space
0 0 1 0 1	S	Bell
0 0 1 1 0	I	8
0 0 1 1 1	U	7
0 1 0 0 0	Car. Ret.	Car. Ret.
0 1 0 0 1	D	\$
0 1 0 1 0	R	4
0 1 0 1 1	J	(Apos)'
0 1 1 0 0	N	(Comma),
0 1 1 0 1	F	!
0 1 1 1 0	C	:
0 1 1 1 1	K	(
1 0 0 0 0	T	5
1 0 0 0 1	Z	"
1 0 0 1 0	L	)
1 0 0 1 1	W	2
1 0 1 0 0	H	Stop
1 0 1 0 1	Y	6
1 0 1 1 0	P	0
1 0 1 1 1	Q	1
1 1 0 0 0	O	9
1 1 0 0 1	B	?
1 1 0 1 0	G	&
1 1 0 1 1	Figures	Figures
1 1 1 0 0	M	.
1 1 1 0 1	X	/
1 1 1 1 0	V	;
1 1 1 1 1	Letters	Letters

Figure 1-15  
5-bit BAUDOT code table.

## Self-Test Review

26. The BCD code is more convenient to use than the binary code because:
- A. it uses less bits.
  - B. it is more compatible with the decimal number system.
  - C. it is more adaptable to arithmetic computations.
  - D. there are more different coding schemes available.
27. Convert the following decimal numbers to 8421 BCD code.
- A. 1049
  - B. 267
  - C. 835
28. Convert the following 8421 BCD code numbers to decimal.
- A. 1001 0110 0010
  - B. 0111 0001 0100 0011
  - C. 1010 1001 1000
  - D. 1000 0000 0101
29. Convert the following binary numbers to 8421 BCD code.
- A. 101110.01
  - B. 1001.0101
  - C. 11011011.0001

30. Convert the following 8421 BCD codes to binary.
- A. 0001 1000 0010.0101
  - B. 0010 1001 0000.0010 0101
  - C. 1101 0110 0011.0101
  - D. 0110 1000.0001 0010 0101
31. Which code is best for error minimizing?
- A. 8421 BCD
  - B. pure binary
  - C. Gray
32. The ASCII and BAUDOT codes are a form of \_\_\_\_\_ codes.
33. To determine if the correct ASCII character has been transmitted, a \_\_\_\_\_ bit is often added to the code.
34. Which type of parity is used when the 8-bit ASCII character 01000111 is transmitted?
- A. odd
  - B. even
35. Refer to Figure 1-12 and convert the following characters into their ASCII 7-bit binary code.
- A. B
  - B. X
  - C. 3
  - D. S
36. Refer to Figure 1-12 and convert the following ASCII 7-bit binary codes to their character equivalents.
- A. 0110010
  - B. 1010110
  - C. 1011010
  - D. 1001110

## Answers

26. B. More compatible with the decimal system.
27. A. 0001 0000 0100 1001  
B. 0010 0110 0111  
C. 1000 0011 0101
28. A. 962  
B. 7143  
C. Invalid (1010 represents a decimal digit greater than 9)  
D. 805
29. A.  $101110.01_2 = (1 \times 2^5) + (0 \times 2^4) + (1 \times 2^3) + (1 \times 2^2) + (1 \times 2^1)$   
 $+ (0 \times 2^0) + (0 \times 2^{-1}) + (1 \times 2^{-2})$   
 $= 32 + 0 + 8 + 4 + 2 + 0 + 0 + 0.25$   
 $= 46.25_{10}$   
 $46.25_{10} = 0100\ 0110.0010\ 0101$   
 $101110.01_2 = 0100\ 0110.0010\ 0101$   
B.  $1001.0101_2 = 1001.0011\ 0001\ 0010\ 0101$   
C.  $11011011.0001_2 = 0010\ 0001\ 1001.0000\ 0110\ 0010\ 0101$

30. A.  $0001\ 1000\ 0010.0101 = 182.5_{10}$   
 $182.5_{10} = 182_{10} + 0.5_{10}$

$182 \div 2 = 91$	with remainder	0	← LSB
$91 \div 2 = 45$		1	
$45 \div 2 = 22$		1	
$22 \div 2 = 11$		0	
$11 \div 2 = 5$		1	
$5 \div 2 = 2$		1	
$2 \div 2 = 1$		0	
$1 \div 2 = 0$		1	← MSB

$182_{10} = 10110110_2$

$0.5 \times 2 = 1.00 = 0$	overflow	1	← MSB
			← LSB

$0.5_{10} = 0.1_2$

$$182.5_{10} = 182_{10} + 0.5_{10} = 10110110_2 + 0.1_2 = 10110110.1_2$$

$$0001\ 1000\ 0010.0101 = 10110110.1_2$$

- B.  $0010\ 1001\ 0000.0010\ 0101 = 100100010.01_2$   
 C. invalid (1101 represents a decimal digit greater than 9)  
 D.  $0110\ 1000.0001\ 0010\ 0101 = 1000100.001_2$

31. Gray

32. Alpha numeric

33. Parity

34. Even

35. A. 1000010  
 B. 1011000  
 C. 0110011  
 D. 1010011

36. A. 2  
 B. V  
 C. Z  
 D. N

## **EXPERIMENTS 1 AND 2**

Perform Experiments 1 and 2 in the Experiment Section, Unit 9 of the course. After you finish the experiment, return to this unit and complete the "Unit Examination."



## UNIT EXAMINATION

This examination will test your knowledge of the important facts in this unit. It will tell you what you have learned and what you need to review. Answer all questions first; then check your work against the correct answers given later.

1. Indicate the base or radix of the following number systems.

- A. Octal \_\_\_\_\_.
- B. Decimal \_\_\_\_\_.
- C. Hexadecimal \_\_\_\_\_.
- D. Binary \_\_\_\_\_.

2. Write the following numbers using positional notation.

- A. 1101.011<sub>2</sub>
- B. 1010.01<sub>10</sub>
- C. 1001.101<sub>8</sub>.
- D. 1110.11<sub>16</sub>

3. Convert the following numbers to decimal.

- A. 10011.011<sub>2</sub>
- B. 752.31<sub>8</sub>
- C. A8C.5F<sub>16</sub>

4. Convert the following numbers to binary.

- A. 105.0625<sub>10</sub>
- B. 374.24<sub>8</sub>
- C. F19.6C<sub>16</sub>

5. Convert the following numbers to octal.

- A. 638.3125<sub>10</sub>
- B. 10010101.0110101<sub>2</sub>

6. Convert the following numbers to hexadecimal.

- A. 9587.03125<sub>10</sub>
- B. 1101101101010.101110101<sub>2</sub>

7. The ASCII and BAUDOT codes are a form of \_\_\_\_\_ codes.
8. Convert the following numbers to 8421 BCD code.
- A.  $521.372_{10}$   
B.  $1010.011_2$
9. Convert the 8421 BCD code 1001 0101.0111 0011 to decimal.
10. Convert the 8421 BCD code 0101 0011.0111 0101 to binary.
11. Which type of parity is used when the 8-bit ASCII character 01110111 is transmitted?
- A. Odd  
B. Even
12. The BAUDOT code uses \_\_\_\_\_ bit numbers to generate a character.
13. Using only your knowledge of binary codes, identify the Gray code.

Decimal	a	b	c	d
0	0000	0000	0000	0011
1	0001	0001	0001	0100
2	0011	0010	0010	0101
3	0010	0011	0011	0110
4	0110	0100	0100	0111
5	0111	0101	1011	1000
6	0101	0110	1100	1001
7	0100	0111	1101	1010
8	1100	1000	1110	1011
9	1101	1001	1111	1100



# Individual Learning Program

## MICROPROCESSORS

*Unit 2*

### MICROCOMPUTER BASICS

EE-3401

HEATH COMPANY  
BENTON HARBOR, MICHIGAN 49022

Copyright © 1977  
Heath Company  
All Rights Reserved  
Printed in the United States of America

## CONTENTS

Introduction .....	2-3
Unit Objectives .....	2-4
Unit Activity Guide .....	2-5
Terms and Conventions .....	2-6
An Elementary Microcomputer .....	2-14
Executing a Program .....	2-28
Addressing Modes .....	2-43
Experiment 3 .....	2-70
Unit Examination .....	2-71
Examination Answers .....	2-76

## *Unit 2*

# MICROPROCESSORS

## INTRODUCTION

A microprocessor is a very complex electronic circuit. It consists of thousands of microscopic transistors squeezed onto a tiny chip of silicon that is often no more than one-eighth inch square. The chip is placed in a package containing up to about 40 leads.

The thousands of transistors that make up the microprocessor are arranged to form many different circuits within the chip. From the standpoint of learning how the microprocessor operates, the most important circuits on the chip are registers, counters, and decoders. In this unit, you will learn how these circuits can work together to perform simple but useful tasks.

## UNIT OBJECTIVES

When you have completed this unit you will be able to:

1. Define the terms: microprocessor, microcomputer, input, output, I/O, I/O device, I/O port, instruction, program, stored program concept, word, byte, MPU, ALU, operand, memory, address, read, write, RAM, fetch, execute, MPU cycle, mnemonic, opcode, and bus.
2. Explain the purpose of the following circuits in a typical microprocessor: accumulator, program counter, instruction decoder, controller sequencer, data register, and address register.
3. Using a simplified block diagram of a hypothetical microprocessor, trace the data flow that takes place between the various circuits during the execution of a simple program.
4. Describe the difference between inherent, immediate, and direct addressing.
5. Write simple, straight-line programs that can be executed by the ET-3400 Microprocessor Trainer.

## UNIT ACTIVITY GUIDE

	Completion Time
<input type="checkbox"/> Read Section on Terms and Conventions.	_____
<input type="checkbox"/> Complete Self-Test Review Questions 1 — 11.	_____
<input type="checkbox"/> Read Section on An Elementary Microcomputer.	_____
<input type="checkbox"/> Complete Self-Test Review Questions 12 — 30.	_____
<input type="checkbox"/> Read Section on Executing a Program.	_____
<input type="checkbox"/> Complete Self-Test Review Questions 31 — 40.	_____
<input type="checkbox"/> Read Section on Addressing Modes.	_____
<input type="checkbox"/> Complete Self-Test Review Questions 41 — 50.	_____
<input type="checkbox"/> Perform Experiment 3.	_____
<input type="checkbox"/> Complete Unit Examination.	_____
<input type="checkbox"/> Check Examination Answers.	_____

## TERMS AND CONVENTIONS

A **microprocessor** is a logic device that is used in digital electronic systems. It is also being used by hobbyists, experimenters and low-budget research groups as a low-cost, general-purpose computer. But a distinction should be made between the microprocessor and the microcomputer.

The microprocessor unit, or MPU, is a complex logic element that performs arithmetic, logic, and control operations. The trend is to package it as a single integrated circuit.

A **microcomputer** contains a microprocessor, but it also contains other circuits such as memory devices to store information, interface adapters to connect it with the outside world, and a clock to act as a master timer for the system. Figure 2-1 shows a typical microcomputer in which these additional circuits are added. The arrows represent conductors over which binary information flows. The wide arrows represent several conductors connected in parallel. A group of parallel conductors which carry information is called a **bus**.

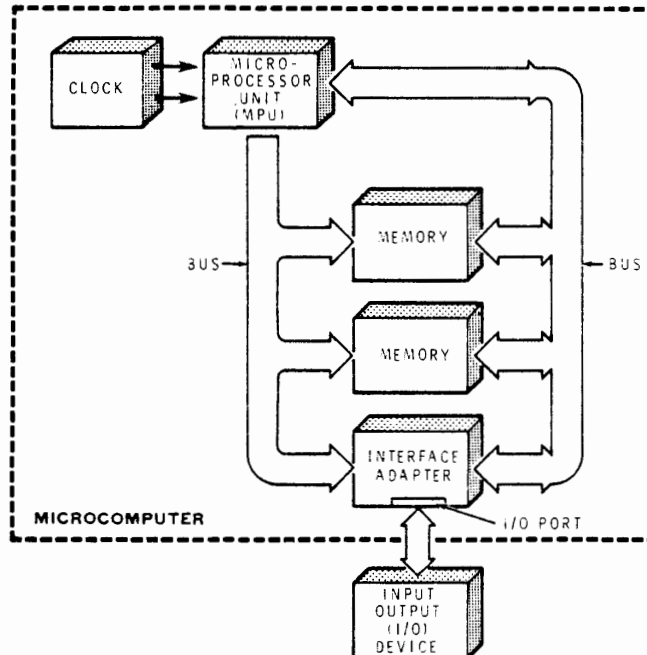


Figure 2-1  
A Basic Microcomputer



The microcomputer is composed of everything inside the dotted line. Everything outside the dotted line is referred to as the **outside world**, and all microcomputers must have some means of communicating with it. Information received by the microcomputer from the outside world is referred to as **input** data. Information transmitted to the outside world from the microcomputer is referred to as **output** data.

Input information may come from devices like paper tape readers, typewriters, mechanical switches, keyboards, or even other computers. Output information may be sent to video displays, output typewriters, paper tape punches, or line printers. Some devices such as the teletypewriter can serve as both an input and an output device. These devices are referred to as **input/output** or **I/O** devices. The point at which the I/O device connects to the microcomputer is called an **I/O port**.

## Stored Program Concept

A microcomputer is capable of performing many different operations. It can add and subtract numbers and it can perform logical operations. It can read information from an input device and transmit information to an output device. In fact, depending on the microprocessor used, there may be 100 or more different operations that the microcomputer can perform. Moreover, two or more individual operations can be combined to perform much more complex operations.

In spite of all its capabilities, the computer will do nothing on its own accord. It will do only what it is told to do, nothing more and nothing less. You must tell the computer exactly what operations to perform and the order in which it should perform them. The operations that the computer can be told to perform are called **instructions**. A few typical instructions are ADD, SUBTRACT, LOAD INDEX REGISTER, STORE ACCUMULATOR, and HALT.

A group of instructions that allow the computer to perform a specific job is called a **program**. One who writes these instructions is called a **programmer**. To design with microprocessors, the engineer must become a programmer. To repair microprocessor-based equipment, the technician must understand programming. Generally, the length of the program is proportional to the complexity of the task that the computer is to perform. A program for adding a list of numbers may require only a dozen instructions. On the other hand, a program for controlling all the traffic lights in a small city may require thousands of instructions.

A computer is often compared to a calculator, which is told what to do by the operator via the keyboard. Even inexpensive calculators can perform several operations that can be compared to instructions in the computer. By depressing the right keys, you can instruct the calculator to add, subtract, multiply, divide, and clear the display. Of course, you must also enter the numbers that are to be added, subtracted, etc. With the calculator, you can add a list of numbers as quickly as you can enter the numbers and the instructions. That is, the operation is limited by the speed and accuracy of the operator.

From the start, computer designers recognized that it was the human operator that slowed the computation process. To overcome this, the **stored program concept** was developed. Using this approach, the program is stored in the computer's memory. Suppose, for example, that you have 20 numbers that are to be manipulated by a program that is composed of 100 instructions. Let's further suppose that 10 answers will be produced in the process.

Before any computation begins, the 100-instruction program plus the 20 numbers are loaded into the computer's memory. Furthermore, 10 memory locations are reserved for the 10 answers. Only then is the computer allowed to execute the program. The actual computation time might be less than one millisecond. Compare this to the time that it would take to manually enter the instructions and numbers, one at a time, while the computer is running. This automatic operation is one of the features that distinguishes the computer from the calculator.

## Computer Words

In computer terminology, a **word** is a group of binary digits that can occupy a storage location. Although the word may be made up of several binary digits, the computer handles each word as if it were a single unit. Thus, the **word** is the fundamental unit of information used in the computer.

A word may be a binary number that is to be handled as data. Or, the word may be an instruction that tells the computer which operation it is to perform. It may be an ASCII character representing a letter of the alphabet. Finally, a word can be an "address" that tells the computer where a piece of data is located.

## Word Length

In the past few years, a wide variety of microprocessors have been developed. Their cost and capabilities vary widely. One of the most important characteristics of any microprocessor is the word length it can handle. This refers to the length in bits of the most fundamental unit of information.

The most common word length for microprocessors is 8 bits. In these units; numbers, addresses, instructions, and data are represented by 8-bit binary numbers. The lowest 8-bit binary number is 0000 0000<sub>2</sub> or 00<sub>16</sub>. The highest is 1111 1111<sub>2</sub> or FF<sub>16</sub>. In decimal, this range is from 0 to 255<sub>10</sub>. Thus, an 8-bit binary number can have any one of 256<sub>10</sub> unique values.

An 8-bit word can specify positive numbers between 0 and 255<sub>10</sub>. Or, if the 8-bit word is an instruction, it can specify any of 256<sub>10</sub> possible operations. It is also entirely possible that the 8-bit word is an ASCII character. In this case, it can represent letters of the alphabet, punctuation marks, or numerals. As you can see, the 8-bit word can represent many different things, depending on how it is interpreted. The programmer must insure that an ASCII character or binary number is not interpreted as an instruction. Later, you will see the consequences of making this mistake.

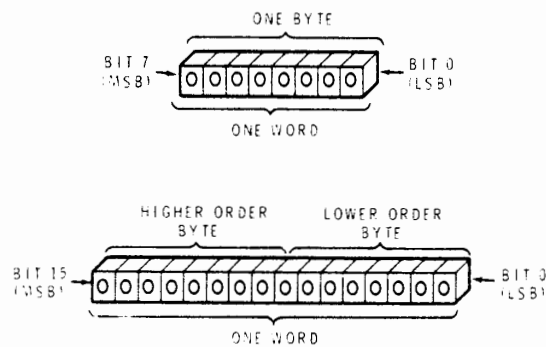
While the 8-bit word length is the most popular, other word lengths are sometimes used. The earliest microprocessor used a 4-bit word length, and four-bit microprocessors are still used in many cases because of their low cost. A few 12-bit and 16-bit microprocessors have also been developed.

Longer word lengths allow us to work with larger numbers. For example, a 16-bit word can represent numbers up to  $65,535_{10}$ . However, this capability adds to the complexity and cost of the microprocessor. Because most microprocessors use 8-bit word lengths, we will restrict our discussion to units of this type.

It should be pointed out that just because the word length is 8-bits, it does not mean that we are restricted to numbers below  $256_{10}$ . It simply means that you must use two or more words to represent larger numbers.

The 8-bit word length defines the size of many different components in the microprocessor system. For example, many of the important registers will have 8-bit capacity. Memory will be capable of holding a large number of 8-bit words. And, the bus which is used to transfer data words will consist of eight parallel conductors.

Even 16-bit microprocessors use 8-bit segments of data in many applications. For example, inputs from teletypewriters often consist of 8-bit ASCII characters. To distinguish these 8-bit segments of information from the 16-bit (or longer) word lengths, another term has come into general use: the term **byte**. A byte is a group of bits that are handled as a single unit. Generally, a byte is understood to consist of 8-bits. In the 8-bit microprocessor, each word consists of one byte. But in the 16-bit machines, each word contains two bytes. Figure 2-2 illustrates these points.



**Figure 2-2**  
Words and Bytes.

Figure 2-2 also shows how the bits that make up the computer word are numbered. The least significant bit (LSB) is on the right while the most significant bit (MSB) is on the left. In the 8-bit word, the bits are numbered 0 through 7 from right to left. In the 16-bit word, the bits are numbered 0 through 15 as shown. The lower 8-bits are called the lower order byte while the upper 8-bits are called the higher order byte.

## Self-Test Review

1. Explain the difference between a microprocessor and a microcomputer.
2. What is a bus?
3. Explain the difference between input and output data.
4. Define I/O.
5. The point at which data enters or leaves the computer is called an I/O\_\_\_\_\_.
6. The operations that the computer can be told to perform are called \_\_\_\_\_.
7. What is a program?
8. Explain what is meant by "stored program concept."
9. A byte generally consists of \_\_\_\_\_ bits.
10. A computer word may consist of one or more \_\_\_\_\_.
11. What is the largest number that can be represented by an 8-bit computer word? By a 16-bit word?

## Answers

1. A microprocessor is a logic element that can perform a variety of arithmetic, control, and logic operations. A microcomputer is a system that consists of a microprocessor, memory, interface adapters, clock, etc.
2. A bus is a group of conductors over which information can be transmitted. The conductors may be wires in a cable, foil patterns on a printed circuit board, or microscopic metal deposits in a silicon chip.
3. Input data is information that is entered into the computer from the outside world. Output data is information that is transmitted from the computer to the outside world. The outside world is defined as anything outside the computer.
4. I/O is the abbreviation for input-output. Thus, a device that can send data to and accept data from the computer is called an I/O device.
5. Port.
6. Instructions.
7. A program is a group of instructions that tell the computer the operations to be performed and the sequence in which they are to be performed.
8. The stored program concept refers to the technique of storing the instruction to be performed in the memory section along with the data that is to be operated upon.
9. 8.
10. Bytes.
11.  $1111\ 1111_2$  or  $255_{10}$ .  $1111\ 1111\ 1111\ 1111_2$  or  $65,535_{10}$ .

## AN ELEMENTARY MICROCOMPUTER

One of the difficulties you may encounter in learning about a microcomputer for the first time is the complexity of its main component — the microprocessor. The microprocessor may have a dozen or more registers varying in size from 1 bit to 16 bits. It will have scores of instructions, most of which can be implemented several different ways. It will have data, address, and control buses. In short, it can be very intimidating to start out by considering a full-blown microprocessor.

A better approach is to start with a “stripped down” version. By initially omitting some of the processor’s advanced features, we arrive at a device that can be readily understood and yet maintains the characteristics of an actual microprocessor. Strictly speaking, the microprocessor developed in this unit is hypothetical in nature. However, it is so close to the real thing that the programs we develop for it will actually run on the ET-3400 Microprocessor Trainer. Also, as you will see later, one of the most popular microprocessors in use today is a vastly advanced version of our elementary model.

A block diagram of a basic microcomputer is shown in Figure 2-3, which shows the microprocessor, the memory, and the I/O circuitry. For simplicity, we will ignore the I/O circuitry in this unit. We can do this by assuming that the program and data are already in memory and that the results of any computations will be held in a register or stored in memory. Ultimately of course, the program and data must come from the outside world and the results must be sent to the outside world. But we will save these procedures until a later unit. This will allow us to concentrate on the microprocessor unit and the memory.

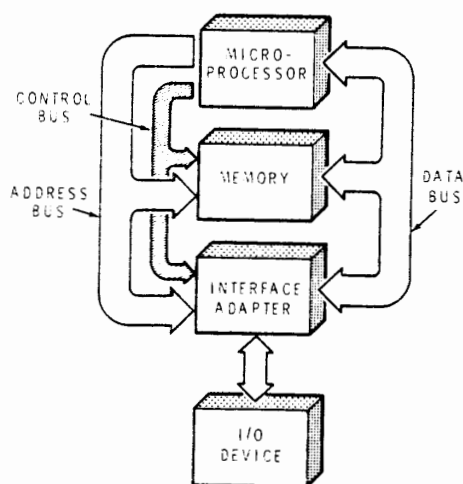


Figure 2-3  
The Basic Microcomputer



## The Microprocessor Unit (MPU)

The microprocessor unit is shown in greater detail in Figure 2-4. For simplicity, only the major registers and circuits are shown. In our elementary unit most of the counters, registers, and buses are 8-bits wide. That is, they can accommodate 8-bit words.

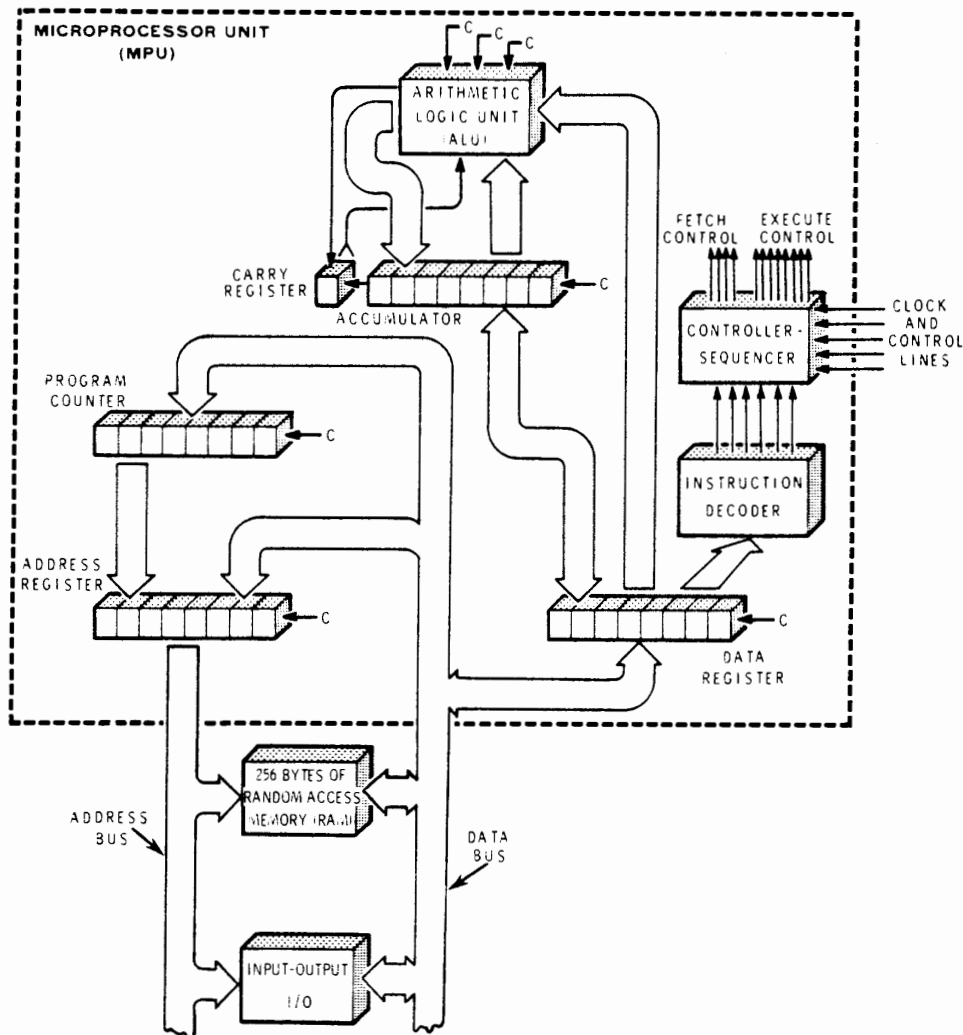


Figure 2-4  
An Elementary Microprocessor.

One of the most important circuits in the microprocessor is the **arithmetic logic unit (ALU)**. Its purpose is to perform arithmetic or logic operations on the data words that are delivered to it. The ALU has two main inputs. One comes from a register called the accumulator, and the other comes from the data register. The ALU can add the two input data words together, or it can subtract one from the other. It can also perform some logic operations which will be discussed in later units. The operation that the ALU performs is determined by signals on the various control lines (marked C on the block diagram).

Generally, the ALU receives two 8-bit binary numbers from the accumulator and the data register as shown in Figure 2-5A. Because some operation is performed on these data words, the two inputs are called **operands**.

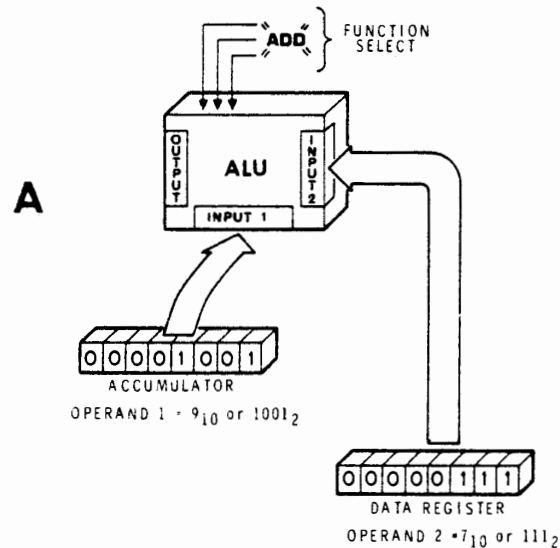
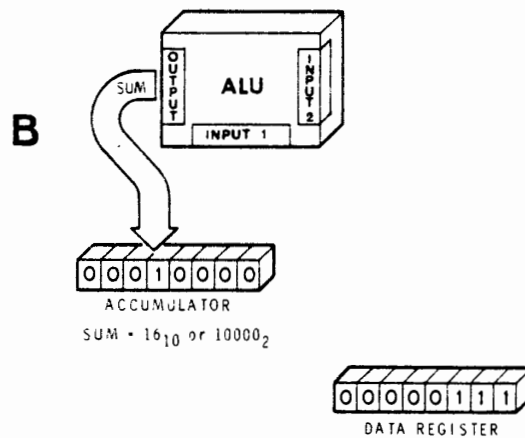


Figure 2-5  
The Arithmetic Logic Unit.



The two operands may be added, subtracted, or compared in some way, and the result of the operation is stored back in the accumulator. For example, assume that two numbers (7 and 9) are to be added. Before the numbers can be added, one operand (9) is placed in the accumulator; the other (7) is placed in the data register. The proper control line is then activated to implement the add operation. The ALU adds the two numbers together, producing their sum ( $16_{10}$ ) at the output. As shown in Figure 2-5B, the sum is stored in the accumulator, replacing the operand that was originally stored there. Notice that all the numbers involved are in binary form.

The **accumulator** is the most useful register in the microprocessor. During arithmetic and logic operations it performs a dual function. Before the operation, it holds one of the operands. After the operation, it holds the resulting sum, difference, or logical answer. The accumulator receives several instructions in every microprocessor. For example, the "load accumulator" instruction causes the contents of some specified memory location to be transferred to the accumulator. The "store accumulator" instruction causes the contents of the accumulator to be stored at some specified location in memory.

The **data register** is a temporary storage location for data going to or coming from the data bus. For example, it holds an instruction while the instruction is being decoded. Also, it holds a data byte while the word is being stored in memory.

The MPU also contains several other important registers and circuits: the address register, the program counter, the instruction decoder, and the controller-sequencer. These are shown in Figure 2-4.

The **address register** is another temporary storage location. It holds the address of the memory location or I/O device that is used in the operation presently being performed.

The **program counter** controls the sequence in which the instructions in a program are performed. Normally, it does this by counting in the sequence, 0, 1, 2, 3, 4, etc. At any given instant, the count indicates the location in memory from which the next byte of information is to be taken.

The **instruction decoder** does just what its name implies. After an instruction is pulled from memory and placed in the data register, the instruction is decoded by this circuit. The decoder examines the 8-bit code and decides which operation is to be performed.

The **controller-sequencer** produces a variety of control signals to carry out the instruction. Since each instruction is different, a different combination of control signals is produced for each instruction. This circuit determines the sequence of events necessary to complete the operation described by the instruction.

Later you will see how these various circuits work together to execute simple programs. But first, take a closer look at the memory for our microcomputer.

## Memory

A simplified diagram of the 256-word, 8-bit read/write memory that is used in our hypothetical microcomputer is shown in Figure 2-6. The memory consists of  $256_{10}$  locations, each of which can store an 8-bit word. This size memory is often referred to as  $256 \times 8$ . A read/write memory is one in which data can be written in and read out with equal ease.

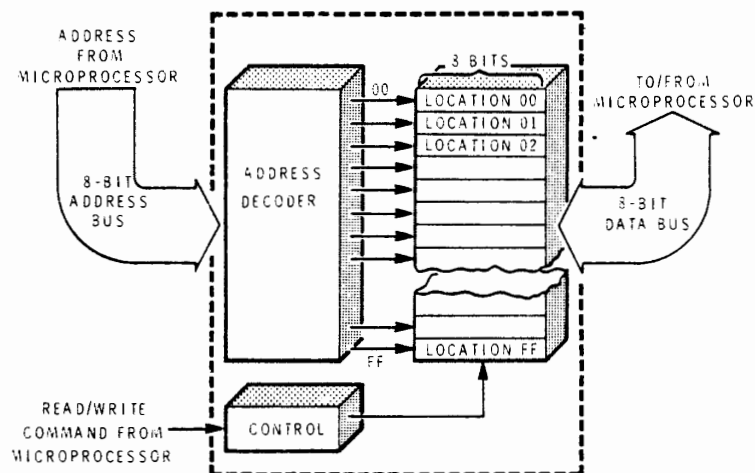


Figure 2-6

The Random Access Memory.

Two buses and a number of control lines connect the memory with the microprocessing unit. The address bus will carry an 8-bit binary number, which can specify  $256_{10}$  locations, from the MPU to the memory address decoder. Each location is assigned a unique number called its address. The first location is given the address 0. The last location is given the address  $255_{10}$ , which is 1111 1111 in binary and FF in hexadecimal. A specific location is selected by placing its 8-bit address on the address bus. The address decoder decodes the 8-bit number and selects the proper memory location.

The memory also receives a control signal from the MPU. This signal tells the memory the operation that is to be performed. A READ signal indicates that the selected location is to be read out. This means that the 8-bit number contained in the selected location is to be placed on the data bus where it can be transferred to the MPU.

The procedure is illustrated in Figure 2-7. Assume that the MPU is to read out the contents of memory location  $04_{16}$ . Let's further assume that the number stored there is  $97_{16}$ . First, the MPU places the address  $04_{16}$  on the address bus. The decoder recognizes the address and selects the proper memory location. Second, the MPU sends a READ signal to the memory, indicating that the contents of the selected location are to be placed on the data bus. Third, the memory responds by placing the number  $97_{16}$  on the data bus. The MPU can then pick up the number and use it as needed.

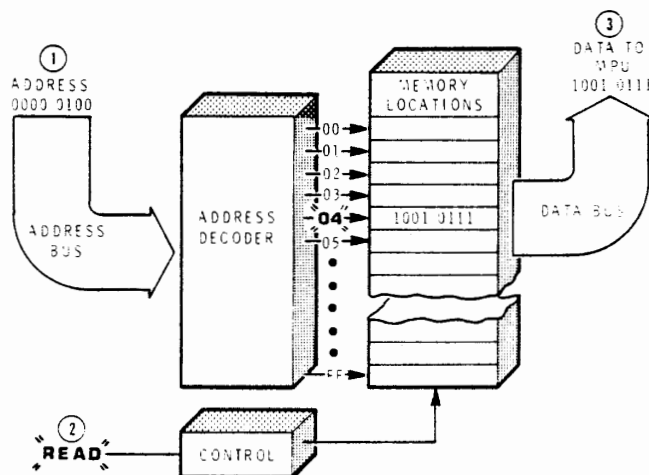


Figure 2-7  
Reading from Memory.

It should be pointed out that the process of reading out a memory location does not disturb the contents of that location. That is, the number  $97_{16}$  will still be present at memory location 04 after the read operation is finished. This characteristic is referred to as nondestructive readout (NDRO). It is an important feature because it allows us to read out the same data as many times as needed.

The MPU can also initiate a WRITE operation. This procedure is illustrated in Figure 2-8. During a WRITE operation, a data word is taken from the data bus and placed in the selected memory location. For example, let's see how the MPU can store the number  $52_{16}$  at memory location 03. First, the MPU places the address 03 on the address bus. The decoder responds by selecting memory location 03. Second, the MPU places the number  $52_{16}$  on the data bus. Third, the MPU sends the WRITE signal. The memory responds by storing the number contained on the data bus in the selected location. That is,  $52_{16}$  is stored in location 03. The previous contents of the selected location are lost as the new number is written in that location.

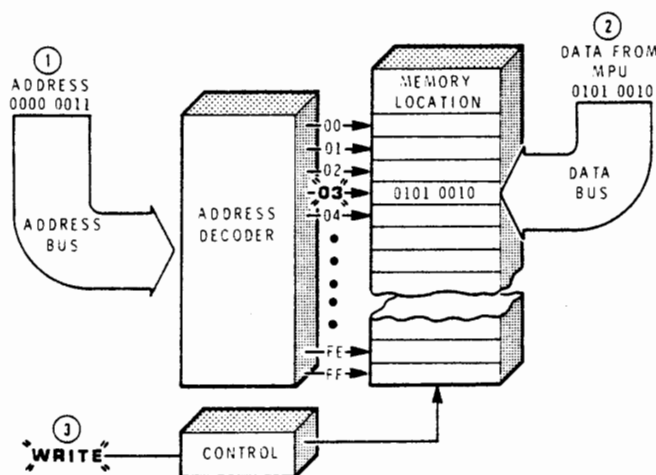


Figure 2-8  
Writing into Memory.

The accepted name for a memory of this type is **Random Access Memory (RAM)**. "Random access" means that all memory locations are equally accessible. However, in recent years RAM has come to mean a random access **read/write** memory. As you will see later, there is another type of memory called a read only memory (ROM). It is also randomly accessible, but it does not have a write capability. Today, the accepted definition of RAM is a random access **read/write** memory. A read only memory, although it is randomly accessible, is called a ROM and never a RAM.

## Fetch-Execute Sequence

When the microcomputer is executing a program, it goes through a fundamental sequence that is repeated over and over again. Recall that a program consists of instructions that tell the microcomputer exactly what operations to perform. These instructions must be stored in an orderly manner in memory. Instructions must be fetched, one at a time, from memory by the MPU. The instruction is then executed by the MPU.

The operation of the microcomputer can be broken down into two phases, as shown in Figure 2-9. When the microprocessor is initially started, it enters the **fetch phase**. During the fetch phase, an instruction is taken from memory and decoded by the MPU. Once the instruction is decoded, the MPU switches to the **execute phase**. During this phase, the MPU carries out the operation dictated by the instruction.

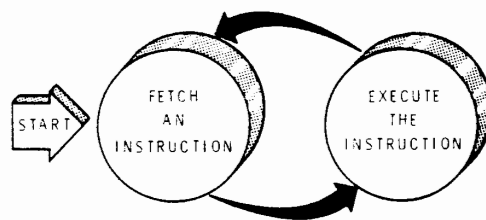


Figure 2-9

The Fetch-Execute Sequence.

The fetch phase always consists of the same series of operations. Thus, it always takes the same amount of time. However, the execute phase will consist of different sequences of events, depending on what type of instruction is being executed. Thus, the time of the execute phase may vary considerably from one instruction to the next.

## A Sample Program

Now that you have a general idea of the registers and circuits found in a microcomputer, we will examine how all these circuits work together to execute a simple program. At this point, we are primarily interested in showing you how the microcomputer operates. Therefore, the program will be a very trivial one.

Let's see how the computer goes about solving a problem like  $7 + 10 = ?$  While this seems like an incredibly easy problem, the computer, if left to its own resources, does not have the foggiest notion of how to solve it. You must tell the computer how to solve the problem right down to the smallest detail. You do this by writing a program.

Before you can write the program you must know what instructions are available to you and the computer. Every microprocessor comes with a listing of its instruction set. Assume that, after looking over the list, you decide that three instructions are necessary to solve the problem. These three instructions and a description of what they do are shown in Figure 2-10.

NAME	MNEMONIC	OPCODE	DESCRIPTION
Load Accumulator	LDA	1000 0110 <sub>2</sub> or 86 <sub>16</sub>	Load the contents of the next memory location into the accumulator.
Add	ADD	1000 1011 <sub>2</sub> or 8B <sub>16</sub>	Add the contents of the next memory location to the present contents of the accumulator. Place the sum in the accumulator.
Halt	HLT	0011 1110 <sub>2</sub> or 3E <sub>16</sub>	Stop all operations.

Figure 2-10

Instructions Used in the Sample Program.

The first column in the table gives the name of the instruction. When writing programs, it is often inconvenient to write out the entire name. For this reason, each instruction is given an abbreviation or a memory aid called a **mnemonic**. The mnemonics are given in the second column. The third column is called the operation code or **opcode**. This is the binary number that the computer and the programmer use to represent the instruction. The opcode is given in both binary and hexadecimal form. The final column describes exactly what operation is performed when the instruction is executed. Study this table carefully; you will be using these instructions over and over again.



Assume that you wish to add 7 to  $10_{10}$  and place the sum in the accumulator. The program is an elementary one. First you will load 7 into the accumulator with the LDA instruction. Next, you will add  $10_{10}$  to the accumulator using the ADD instruction. Finally, you will stop the computer with the HLT instruction.

Using the mnemonics and the decimal representation of the numbers to be added, the program looks like this:

```
LDA 7
ADD 10
HLT
```

Unfortunately, the basic microcomputer cannot understand mnemonics or decimal numbers. It can interpret binary numbers and nothing else. Thus, you must write the program as a sequence of binary numbers. You can do this by replacing each mnemonic with its corresponding opcode and each decimal number with its binary counterpart.

That is:

LDA 7	becomes	1000 0110	0000 0111
		opcode from	binary representation
		Figure 2-10	for 7

And:

ADD 10	becomes	1000 1011	0000 1010
		opcode from	binary representation
		Figure 2-10	for $10_{10}$

Finally,

HLT	becomes	0011 1110
		opcode from
		Figure 2-10

Notice that the program consists of three instructions. The first two instructions have two parts: an 8-bit opcode followed by an 8-bit operand. The operands are the two numbers that are to be added (7 and  $10_{10}$ ).

Recall that the microprocessor and memory work with 8-bit words or bytes. Because the first two instructions consist of 16-bits of information, they must be broken into two 8-bit bytes before they can be stored in memory. Thus, when the program is stored in memory, it will look like this:

1st Instruction	{	1000 0110	Opcode for LDA
		0000 0111	Operand (7)
2nd Instruction	{	1000 1011	Opcode for ADD
		0000 1010	Operand (10 <sub>10</sub> )
3rd Instruction		0011 1110	Opcode for HLT

Five bytes of memory are required. You can store this 5-byte program any place in memory you like. Assuming you store it at the first five memory locations, the memory can be diagrammed as shown in Figure 2-11.

ADDRESS		MEMORY	MNEMONICS/CONTENTS
HEX	BINARY	BINARY CONTENTS	
00	0000 0000	1 0 0 0 0 1 1 0	LDA
01	0000 0001	0 0 0 0 0 1 1 1	7
02	0000 0010	1 0 0 0 1 0 1 1	ADD
03	0000 0011	0 0 0 0 1 0 1 0	10 <sub>10</sub>
04	0000 0100	0 0 1 1 1 1 1 0	HLT
	.		
	.		
	.		
	.		
	.		
FD	1111 1101		
FE	1111 1110		
FF	1111 1111		

Figure 2-11

The Program in Memory.

Notice that each memory location has two 8-bit binary numbers associated with it. One is its address, the other is its contents. Be careful not to confuse these two numbers. The address is fixed. It is established when the microcomputer is built. However, the contents may be changed at any time by storing new data.

Before you see how this program is executed, let's review the material covered in this section.

## Self-Test Review

12. The circuit in the microprocessor that performs arithmetic and logic operations is called the \_\_\_\_\_.
13. The numbers that are operated upon by the computer are called \_\_\_\_\_.
14. Where are the two operands held as they are transferred to the ALU?
15. Where is the result held after an arithmetic operation?
16. Which register in the MPU holds the instruction while the opcode is being decoded?
17. Which circuit in the MPU determines the memory location from which the next byte of information will be taken?
18. Which register holds the address while it is being decoded?
19. How many memory locations can be specified by an 8-bit address?
20. Explain the 3-step procedure for writing the number  $34_{16}$  into memory location 9.
21. Define mnemonic.
22. Define opcode.
23. Define RAM.
24. An instruction is retrieved from memory and decoded during the \_\_\_\_\_ phase.
25. The operation indicated by the instruction is carried out during the \_\_\_\_\_ phase.
26. In our hypothetical microcomputer, the mnemonic for the load accumulator instruction is \_\_\_\_\_.
27. The opcode for the halt instruction is \_\_\_\_\_.
28. When the ADD instruction is executed, where is the SUM stored?

29. How many memory locations are required to store the following program?

LDA  $13_{10}$   
ADD  $17_{10}$   
ADD  $10_{10}$   
HLT

30. What will be the contents of the accumulator after this program is executed?

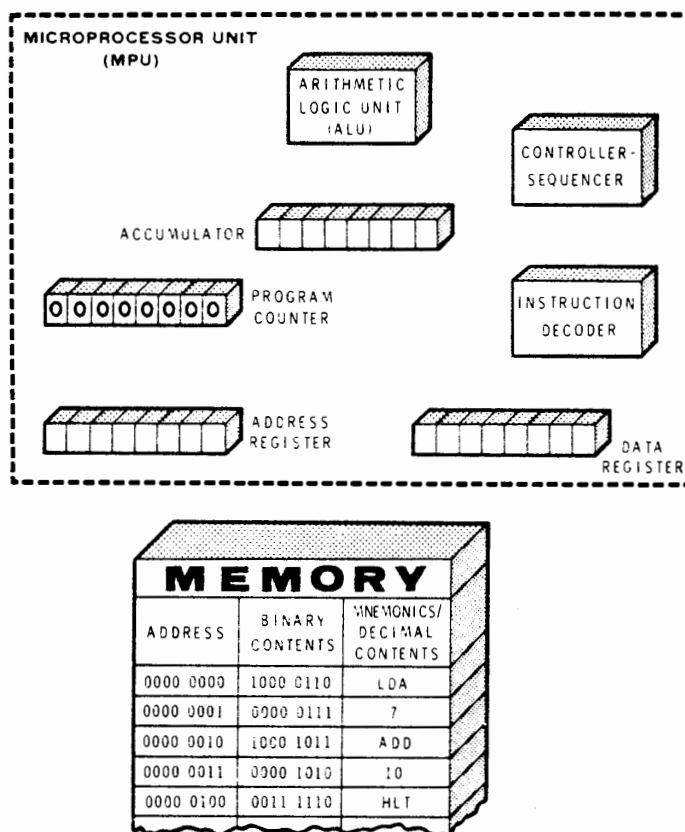
## Answers

12. Arithmetic logic unit (ALU).
13. Operands.
14. One is held in the accumulator; the other in the data register.
15. In the accumulator.
16. Data register.
17. The program counter.
18. The address register.
19.  $2^8 = 256_{10}$ .
20. Step 1. The address 9 is placed on the address bus.  
Step 2. The data  $34_{16}$  is placed on the data bus.  
Step 3. The read/write line is switched to the write state.
21. A mnemonic is an abbreviation or memory aid.
22. An opcode is a binary number that tells the microprocessor which instruction to execute.
23. RAM has come to mean a random access read/write memory.
24. Fetch.
25. Execute.
26. LDA.
27.  $0011\ 1110_2$ .
28. In the accumulator.
29. Seven.
30.  $40_{10}$  or  $101000_2$ .

## EXECUTING A PROGRAM

Before a program can be run, it must be placed in memory. Later, you will see how this is done. For now, assume that we have already loaded the program developed in the previous section.

The important registers of the microcomputer are shown in Figure 2-12. Notice that our 5-byte program for adding 7 and 10<sub>10</sub> is shown in memory. The following paragraphs will take you through the step-by-step procedure by which the computer executes this program.



**Figure 2-12**  
The Program Counter is Set to the Address of the First Instruction.

To begin executing the program, the program counter must be set to the address of the first instruction. In this case, the first instruction is in memory location 0000 0000, so the program counter is set accordingly. The procedure for setting the program counter to the proper address will be discussed later.

## The Fetch Phase

The first step is to fetch the first instruction from memory. The sequence of events that happen during the fetch phase is controlled by the controller-sequencer. It produces a number of control signals which will cause the events illustrated in Figure 2-13 through 2-17 to occur.

First the contents of the program counter are transferred to the address register as shown in Figure 2-13. Recall that this is the address of the first instruction.

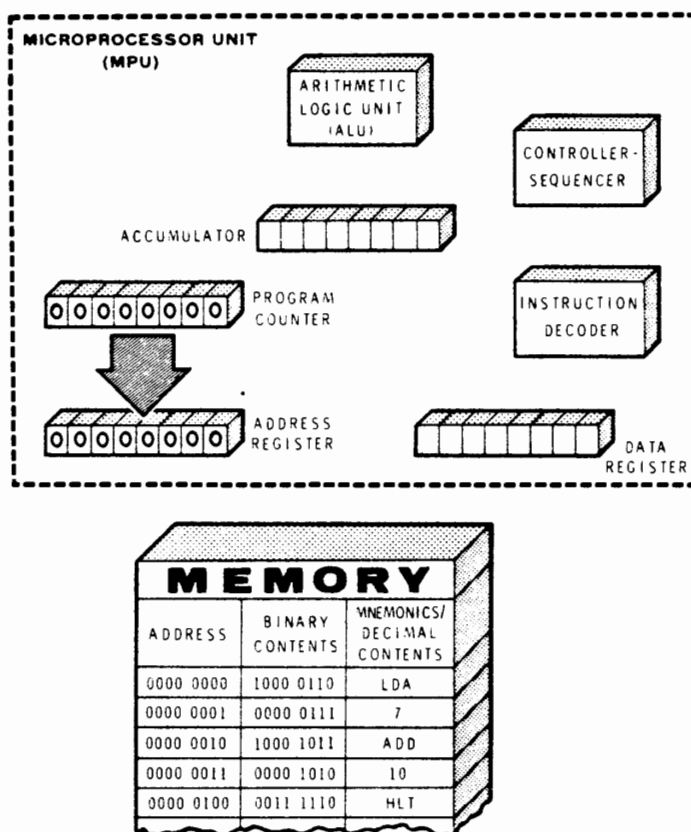
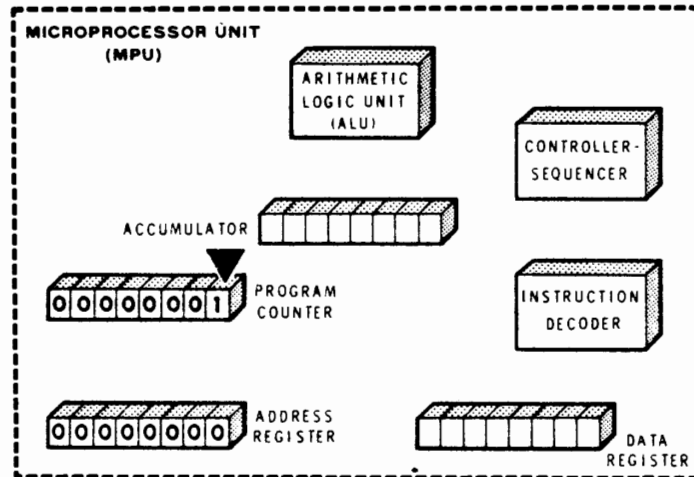


Figure 2-13  
The Contents of the Program Counter  
are Transferred to the Address Register.

Once the address is safely in the address register, the program counter is incremented by one. That is, its contents change from 0000 0000 to 0000 0001 (Figure 2-14). Notice that this does not change the contents of the address register in any way.

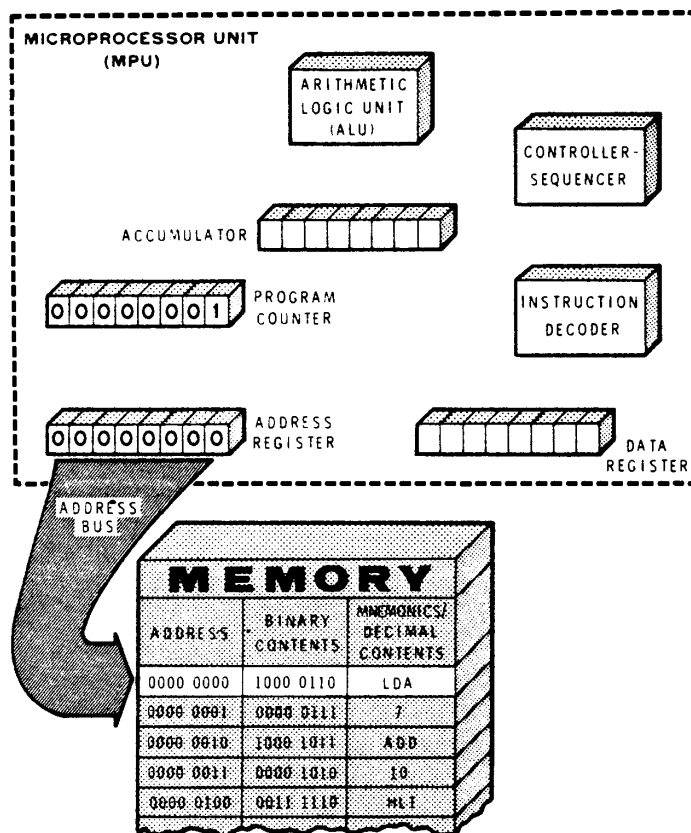


MEMORY		
ADDRESS	BINARY CONTENTS	MNEMONICS/DECIMAL CONTENTS
0000 0000	1000 0110	LDA
0000 0001	0000 0111	7
0000 0010	1000 1011	ADD
0000 0011	0000 1010	10
0000 0100	0011 1110	HLT

Figure 2-14  
The Program Counter is Incremented.



Next, the contents of the address register (0000 0000) are placed on the address bus (Figure 2-15). The memory circuits decode the address and select memory location 0000 0000.



**Figure 2-15**  
The Address of the First Instruction is  
Placed on the Address Bus.

The contents of the selected memory location are placed on the data bus and transferred to the data register in the MPU. After this operation, the opcode for the LDA instruction will be in the data register as shown in Figure 2-16.

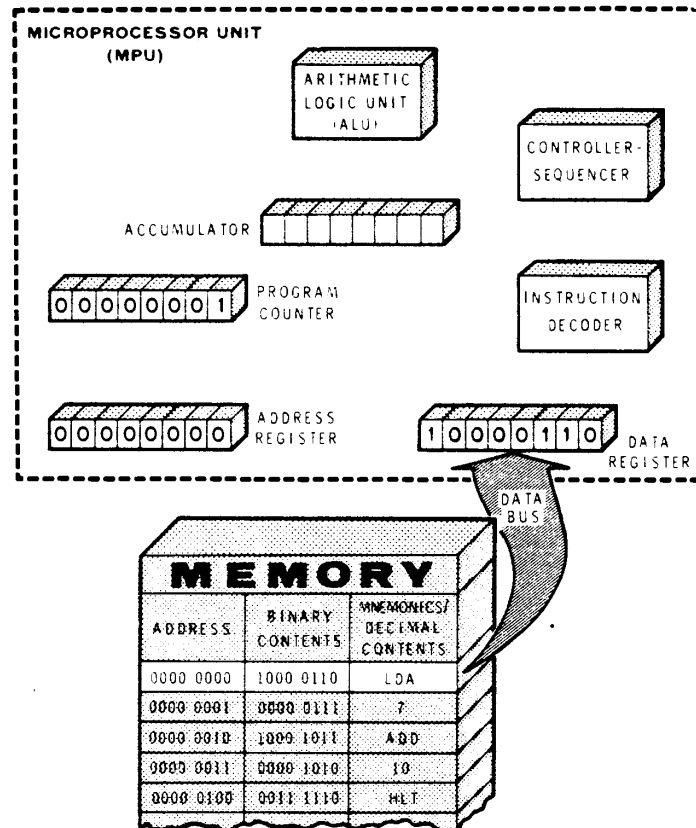


Figure 2-16  
The Opcode of the First Instruction is  
Placed on the Data Bus.

The next step is to decode the instruction (Figure 2-17). The opcode is transferred to the instruction decoder. This circuit recognizes that the opcode is that of an LDA instruction. It informs the controller-sequencer of this fact and the sequencer produces the necessary control pulses to carry out the instruction. This completes the fetch phase of the first instruction.

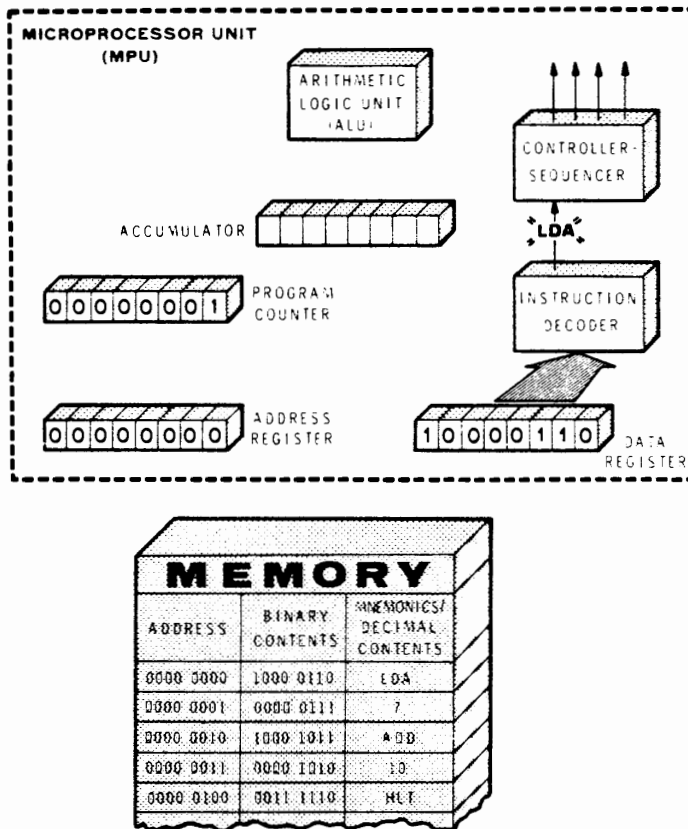


Figure 2-17  
The Opcode is Decoded.

## The Execute Phase

The first instruction was fetched from memory and decoded during the fetch phase. The MPU now knows that this is an LDA instruction. During the execute phase, it must carry out this instruction by reading out the next byte of memory and placing it in the accumulator.

The first step is to transfer the address of the next byte from the program counter to the address register (Figure 2-18). You will recall that the program counter was incremented to the proper address (0000 0001) during the previous fetch phase.

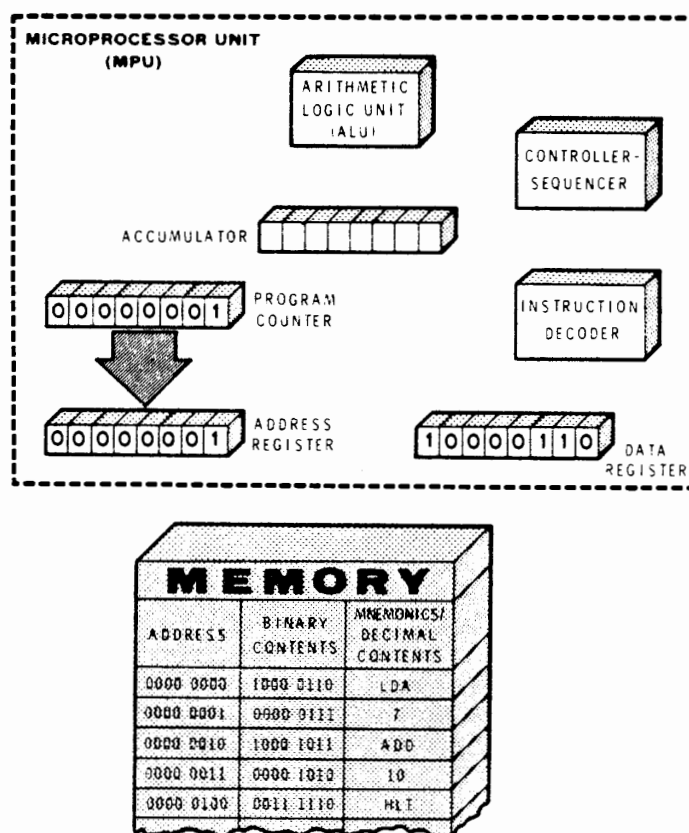
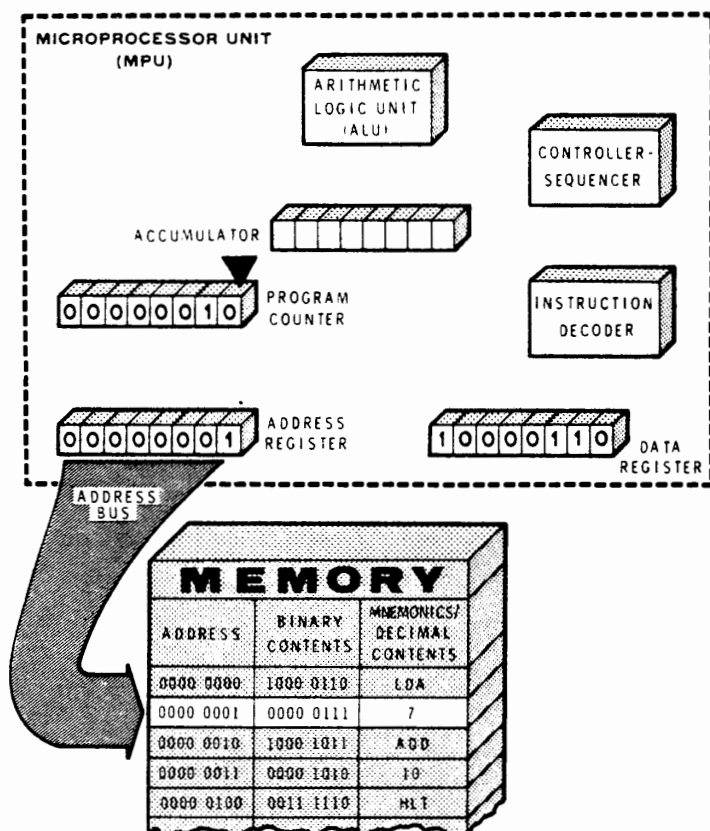


Figure 2-18

The Contents of the Program Counter are Transferred to the Address Register.

The next two operations are shown in Figure 2-19. First, the program counter is incremented to 0000 0010 in anticipation of the next fetch phase. Second, the contents of the address register (000 0001) are placed on the address bus.



**Figure 2-19**  
The Program Counter is Incremented;  
the Contents of the Address Register  
are Placed on the Address Bus.

The address is decoded and the contents of memory location 0000 0001 are loaded into the data register as shown in Figure 2-20. Recall that this is the number 7. An instant later, the number is transferred to the accumulator. Thus, the first execute phase ends with the number 7 in the accumulator.

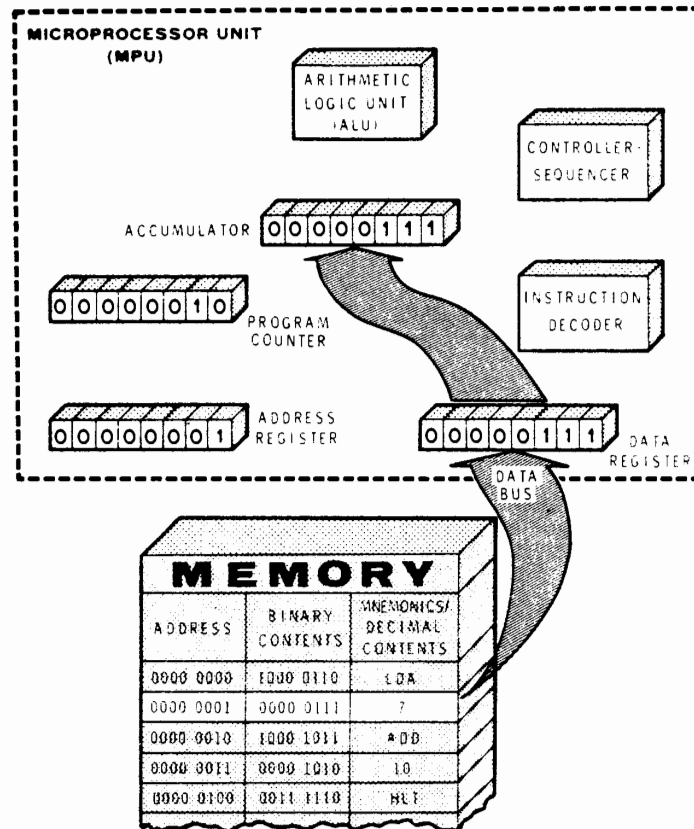


Figure 2-20  
The First Operand is Transferred to the Accumulator Via the Data Register.

## Fetching the Add Instruction

The next instruction in our program is the ADD instruction. It is fetched from memory using the same procedure outlined above. Figure 2-21 illustrates this five-step procedure:

1. The contents of the program counter (0000 0010) are transferred to the address register.

2. The program counter is incremented to 0000 0011.
3. The address is placed on the address bus.
4. The contents of the selected memory location are transferred to the data register.
5. The contents of the data register are decoded by the instruction decoder.

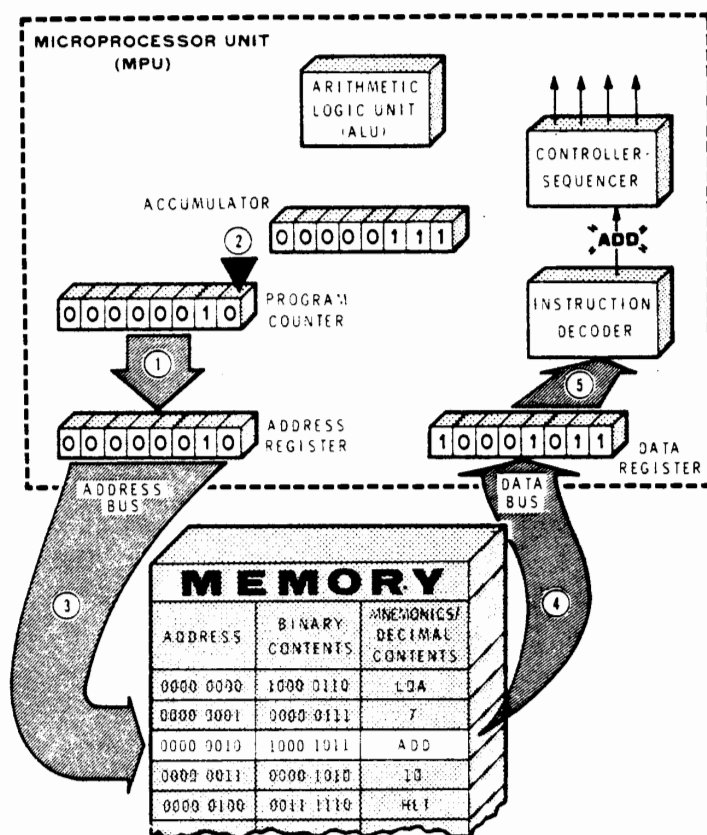


Figure 2-21  
 Fetching the ADD Instruction.

The data word fetched from memory is the opcode for the ADD instruction. Thus, the controller-sequencer produces the necessary control pulses to execute this instruction.

## Executing the Add Instruction

The execution of the ADD instruction is a six-step procedure. This procedure is illustrated in Figure 2-22.

1. The contents of the program counter (0000 0011) are transferred to the address register.
2. The program counter is incremented to 0000 0100 in anticipation of the next fetch phase.
3. The address of the operand is placed on the address bus.
4. The operand ( $10_{10}$ ) is transferred to the data register.
- 5A. The operand ( $10_{10}$ ) is transferred into one input of the ALU.
- 5B. Simultaneously, the other operand (7) is transferred from the accumulator to the other input of the ALU.
6. The ALU adds the two operands. Their sum ( $0001\ 0001_2$  or  $17_{10}$ ) is loaded into the accumulator, destroying the number (7) that was previously stored there.



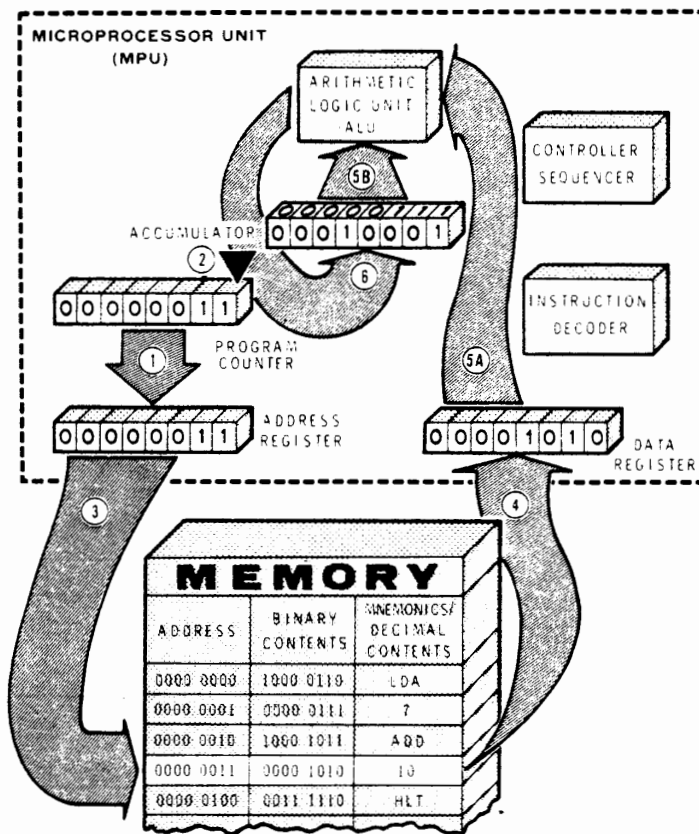


Figure 2-22  
Executing the ADD Instruction.

The computation portion of our program ends with the sum of the two operands in the accumulator. However, the program is not finished until it tells the computer to stop executing instructions.

## Fetching and Executing the HLT Instruction

The final instruction in the program is a HLT instruction. It is fetched using the same fetch procedure as before. The five steps are illustrated in Figure 2-23. The address comes from the program counter via the address register. When this address is placed on the address bus, memory location 0000 0100 is read out and the opcode for HLT is loaded into the data register. The opcode is decoded and the instruction is executed.

The execution of the HLT instruction is very simple. The controller-sequencer simply stops producing control signals. Consequently, all computer operations stop. Notice that the program has accomplished our objective of adding  $7_{10}$  to  $10_{10}$ . The resulting sum,  $17_{10}$ , is in the accumulator.

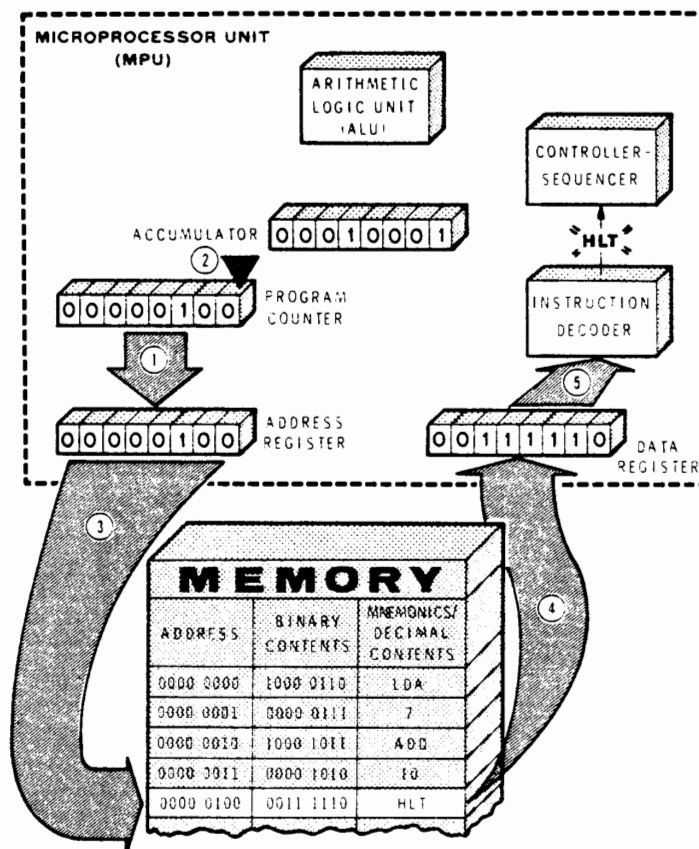


Figure 2-23  
Fetching and Executing the HLT Instruction.

## Self-Test Review

Examine this sample program carefully and answer the questions below:

```
LDA 8  
ADD 7  
ADD 4  
HLT
```

31. During the first fetch phase, what binary number is loaded into the data register?
32. During the first execute phase the number 0000 1000<sub>2</sub> is loaded into the \_\_\_\_\_.
33. During the second fetch phase, what binary number is loaded into the data register?
34. If the first byte of the program is placed in address 0000 0000, what is the address of the first ADD instruction?
35. How many bytes of memory are taken up by the program?
36. What number is in the accumulator during the third fetch phase?
37. When the program is finished running, what number is in the accumulator?
38. What is the final number in the program counter?
39. What are the final contents of the address register?
40. What are the final contents of the data register?

## Answers

31. The opcode for the LDA instruction,  $1000\ 0110_2$ .
32. Accumulator.
33. The opcode for ADD,  $1000\ 1011$ .
34.  $0000\ 0010_2$ .
35. Seven.
36.  $15_{10}$  or  $0000\ 1111_2$ .
37.  $19_{10}$  or  $0001\ 0011_2$ .
38.  $7_{10}$  or  $0000\ 0111_2$ .
39.  $6_{10}$  or  $0000\ 0110_2$ .
40. The opcode for the HLT instruction,  $0011\ 1110_2$ .

## ADDRESSING MODES

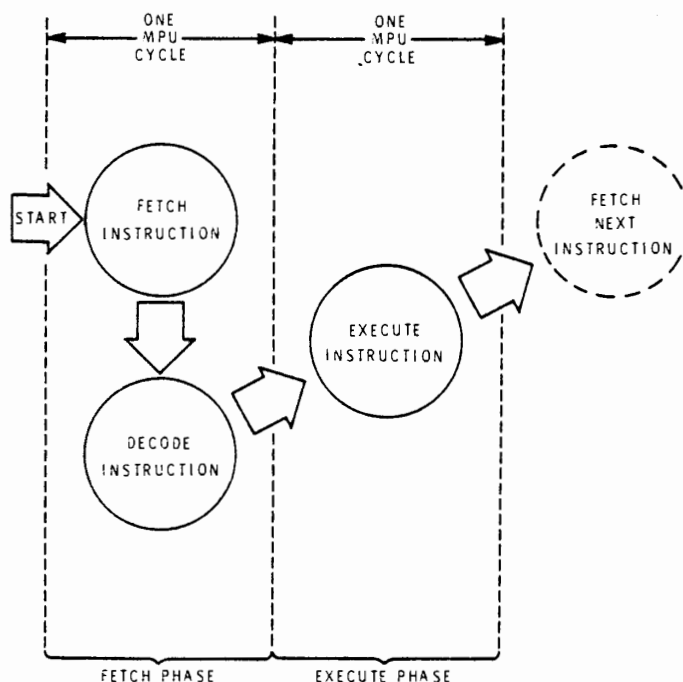
If you examine the program discussed in the previous section, you will find that it uses two distinctly different types of instructions. One type of instruction requires an operand. LDA and ADD are examples of this type. These are two-byte instructions. The first byte is the opcode; the second is the operand.

Microprocessors also have single-byte instructions. HLT is a good example. This instruction requires no operand; thus, it can be implemented with a single byte.

Instructions can be classified in several different ways. One of the most basic distinctions is their addressing mode. The addressing mode refers to the method by which an instruction addresses its operand.

## Inherent or Implied Addressing

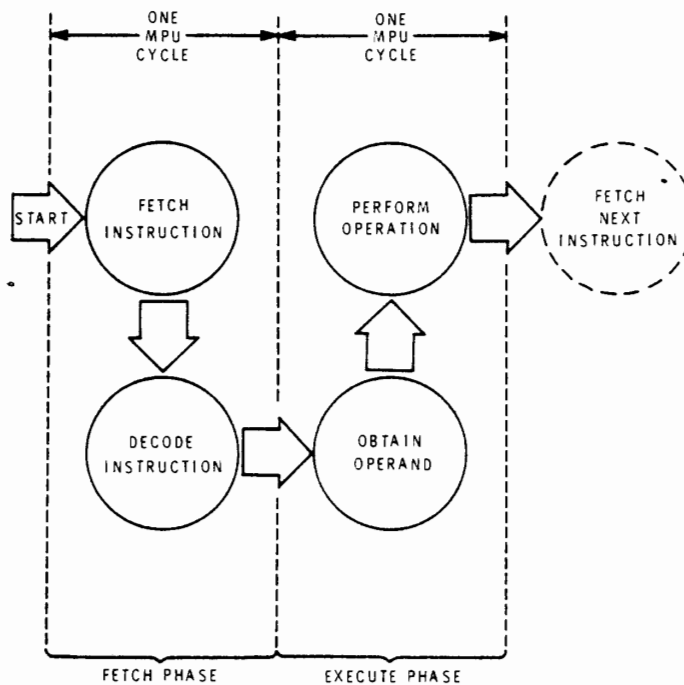
Single-byte instructions have no operand or the operand is implied by the opcode itself. For example, the HLT instruction has no operand at all. However, some single-byte instructions may have an implied operand. An example is the “increment accumulator” instruction. The number that is operated upon (incremented) is the number in the accumulator. This instruction simply adds one to the contents of the accumulator. This type of addressing will be referred to in this course as **implied addressing** or **inherent addressing**. The operations performed during an instruction of this type are illustrated in Figure 2-24.



**Figure 2-24**  
Operations Performed in the Inherent  
or Implied Addressing Mode.

## Immediate Addressing

In our previous program, the two-byte instructions use the **immediate addressing** mode. In this mode, the operand is the byte immediately following the opcode. That is, the byte of data that is to be operated upon is the second byte of the instruction. When these two-byte instructions are stored in memory, the address of the operand is the memory location following the opcode. The operations performed during this type of instruction are illustrated in Figure 2-25.



**Figure 2-25**  
Operations Performed in the Im-  
mediate Addressing Mode.

The inherent and immediate addressing modes have two advantages. First, they require little memory space; one and two bytes respectively. This is important because memory locations cost money. Generally, the less memory space taken by a program, the better off we are. Second, these addressing modes require a minimum of execution time. The execution time can become important in long programs.

The time required for an instruction to be fetched and executed is often given in **MPU cycles**. We will define an MPU cycle as the minimum time required to fetch a data byte from memory. Thus, the fetch phase of an instruction requires one MPU cycle. In inherent and immediate addressing modes, the execute phase also requires one MPU cycle. Therefore, the **minimum** time required to fetch and execute any instruction is two MPU cycles. As you will see later, other addressing modes will require more MPU cycles.

Because of their fast execution time and their efficient use of memory, the inherent and immediate modes of addressing should be used wherever possible. However, there are many situations in which these addressing modes are simply not suitable. For this reason, every microprocessor will have instructions which use other addressing modes.

## Direct Addressing

Most computer operations involve an operand. To realize its full potential, the computer must be able to manipulate the operand in many different ways. Immediate addressing of an operand is most useful when the operand is a constant that is used by only one instruction in the program. In this case, the operand can be placed immediately after that instruction's opcode.

However, there are many cases in which the operand is a variable which may be operated upon by many different instructions. In these cases, the immediate addressing mode is simply not practical and a more sophisticated form of addressing is necessary.



One way of solving this problem is to use the **direct addressing mode**. In this mode, an instruction requires two bytes of memory. The first byte is the opcode of the instruction just as before. However, the second byte is **not** the operand. Instead, the second byte is the **address** of the operand. The format of the instruction as it appears in memory is shown in Figure 2-26.

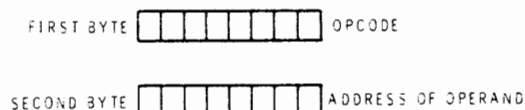


Figure 2-26

Format of an Instruction which uses  
the Direct Addressing Mode.

Three typical direct-addressing-mode instructions are listed in Figure 2-27. The first is the load accumulator instruction (LDA). Read the description carefully and note the difference between this instruction and the LDA immediate instruction discussed earlier. An example of each may help. In the **immediate addressing mode**, the instruction

LDA 50<sub>10</sub>

means "load 50<sub>10</sub> into the accumulator." But in the **direct addressing mode**, the same instruction means "load the number at memory location 50<sub>10</sub> into the accumulator."

NAME	MNEMONIC	OPCODE	DESCRIPTION
Load Accumulator	LDA	1001 0110 <sub>2</sub> or 96 <sub>16</sub>	Load the contents of the memory location whose address is given by the next byte into the accumulator.
Add	ADD	1001 1011 <sub>2</sub> or 9B <sub>16</sub>	Add the contents of the memory location whose address is given by the next byte to the present contents of the accumulator. Place the sum in the accumulator.
Store Accumulator	STA	1001 0111 <sub>2</sub> or 97 <sub>16</sub>	Store the contents of the accumulator in the memory location whose address is given by the next byte.

Figure 2-27

Direct Addressing Mode Instructions.

You may have noticed that this instruction has a different opcode from the LDA immediate instruction. This is necessary to tell the MPU the exact nature of the instruction. That is, the opcode tells the MPU the addressing mode as well as the operation that is to be performed.

The ADD instruction also has a slightly different meaning in the direct addressing mode. Recall that, in the **immediate** addressing mode,

ADD  $10_{10}$

means "add  $10_{10}$  to the contents of the accumulator." However, in the **direct** addressing mode,

ADD  $10_{10}$

means "add the contents of memory location  $10_{10}$  to the contents of the accumulator." Once again, the opcode tells the MPU the addressing mode. An opcode of  $8B_{16}$  means ADD immediate whereas an opcode of  $9B_{16}$  means ADD direct.

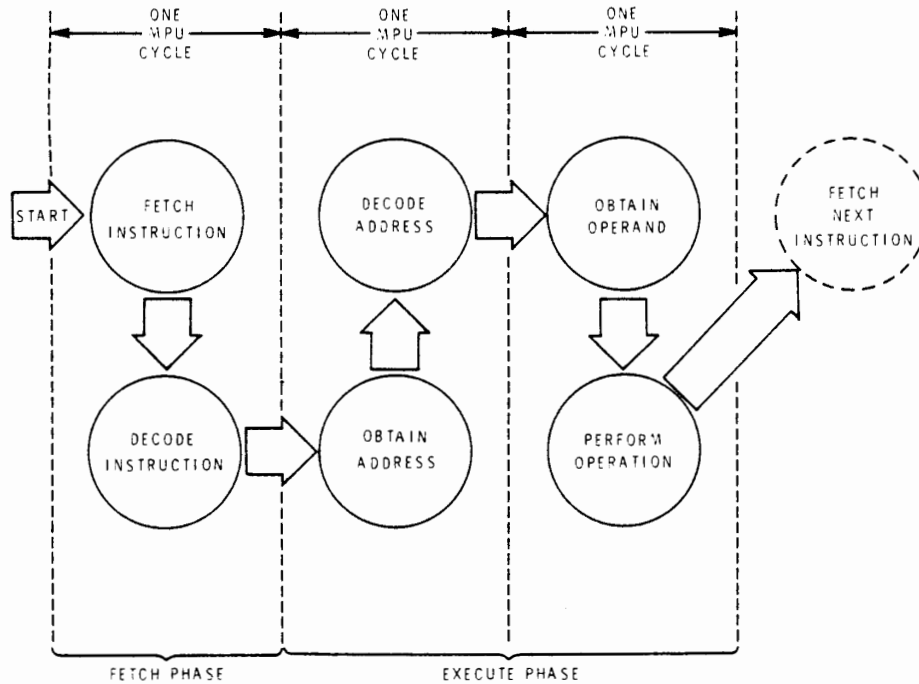
The last instruction shown is a store accumulator (STA) instruction. It tells the MPU to store the contents of the accumulator in the address indicated by the second byte of the instruction.

For example:

STA  $20_{10}$

means "store the contents of the accumulator in memory location  $20_{10}$ ." Because the second byte of the instruction must be an address, there is no STA immediate instruction.

The direct addressing mode instructions require one or more additional MPU cycles to execute. A typical fetch-execute sequence is illustrated in Figure 2-28. The instruction fetch is the same regardless of the addressing mode. However, the execution phase is extended since the MPU must first obtain the address of the operand from memory and then retrieve the operand itself from memory.



**Figure 2-28**  
Most Direct Addressing Mode Instruc-  
tions Require Three MPU Cycles.

Direct addressing generally requires more memory and longer execution times. However, there are many cases in which the added flexibility makes direct addressing worthwhile in spite of these disadvantages.

## Sample Program Using Direct Addressing

The differences between direct and immediate addressing can be illustrated by a sample program. Earlier we examined a program which added two numbers (7 and 10). The sum was placed in the accumulator. Now let's look at the same program using direct addressing, only this time the sum will be stored in memory.

Figure 2-29 shows the program as it would look when stored in memory. Assume that we have arbitrarily stored the program starting at memory location or address  $0001\ 0000_2$  ( $16_{10}$ ). Addresses  $16_{10}$  and  $17_{10}$  contain the first instruction:

LDA  $23_{10}$

BINARY ADDRESS	BINARY CONTENTS	MNEMONICS/ CONTENTS
0001 0000	1001 0110	LDA
0001 0001	0001 0111	$23_{10}$ } 1st Instruction
0001 0010	1001 1011	ADD
0001 0011	0001 1000	$24_{10}$ } 2nd Instruction
0001 0100	1001 0111	STA
0001 0101	0001 1001	$25_{10}$ } 3rd Instruction
0001 0110	0011 1110	HLT
0001 0111	0000 0111	$7_{10}$ } 4th Instruction
0001 1000	0000 1010	$10_{10}$ } Data
0001 1001	0000 0000	Reserved for sum

Figure 2-29  
Sample Program Using Direct Addressing.

This instruction tells the MPU to load the contents of memory location  $23_{10}$  into the accumulator. Looking down to address  $23_{10}$  ( $0001\ 0111_2$ ), you see that it contains the operand 7. Thus, the first instruction causes 7 to be loaded into the accumulator.

The second instruction is in memory locations  $18_{10}$  and  $19_{10}$ . It is:

ADD  $24_{10}$

This tells the MPU to add the number at address  $24_{10}$  to the number in the accumulator. Address  $24_{10}$  ( $0001\ 1000_2$ ) contains the number  $10_{10}$ . Therefore, the second instruction causes  $10_{10}$  to be added to the contents of the accumulator. The sum ( $17_{10}$ ) is placed back in the accumulator.

The third instruction, in locations  $20_{10}$  and  $21_{10}$ , is:

STA  $25_{10}$

This tells the MPU to store the contents of the accumulator in memory location  $25_{10}$ . After this instruction is executed, the number  $17_{10}$  will appear in location  $25_{10}$ .

The final instruction tells the MPU to halt. The program illustrates the value of the HLT instruction. Let's assume that the HLT instruction is inadvertently omitted. In this case, the MPU would fetch the next byte in sequence and attempt to execute it as if it were an instruction. The next byte is the number  $7_{10}$ . This is a data word and was never intended to be an instruction. Nevertheless, the MPU probably has an instruction with an opcode of  $7_{10}$ . After executing this instruction, the MPU will continue to fetch and execute whatever it finds in the remaining memory locations. Without a HLT instruction, it has no way of knowing where the instructions end and data begins.

## Executing the Sample Program

The data flow within the microcomputer is slightly different for the direct addressing mode. Figure 2-30 shows several of the data paths within the microcomputer. Using this type of diagram, let's examine the data manipulations that occur during the execution of our sample program.

Notice that our program is loaded in memory starting at address  $16_{10}$ . The program counter is set to  $16_{10}$ , so the MPU is ready to begin executing the program.

The first fetch phase is illustrated in Figure 2-30. During this phase:

1. The contents of the program counter are loaded into the address register.
2. The program counter is incremented to  $17_{10}$ .
3. The contents of the address register are placed on the address bus.
4. The contents of the selected memory location are transferred via the data bus to the data register.
5. The contents of the data register are decoded.
6. The MPU recognizes that an LDA direct operation is indicated. This concludes the fetch phase.

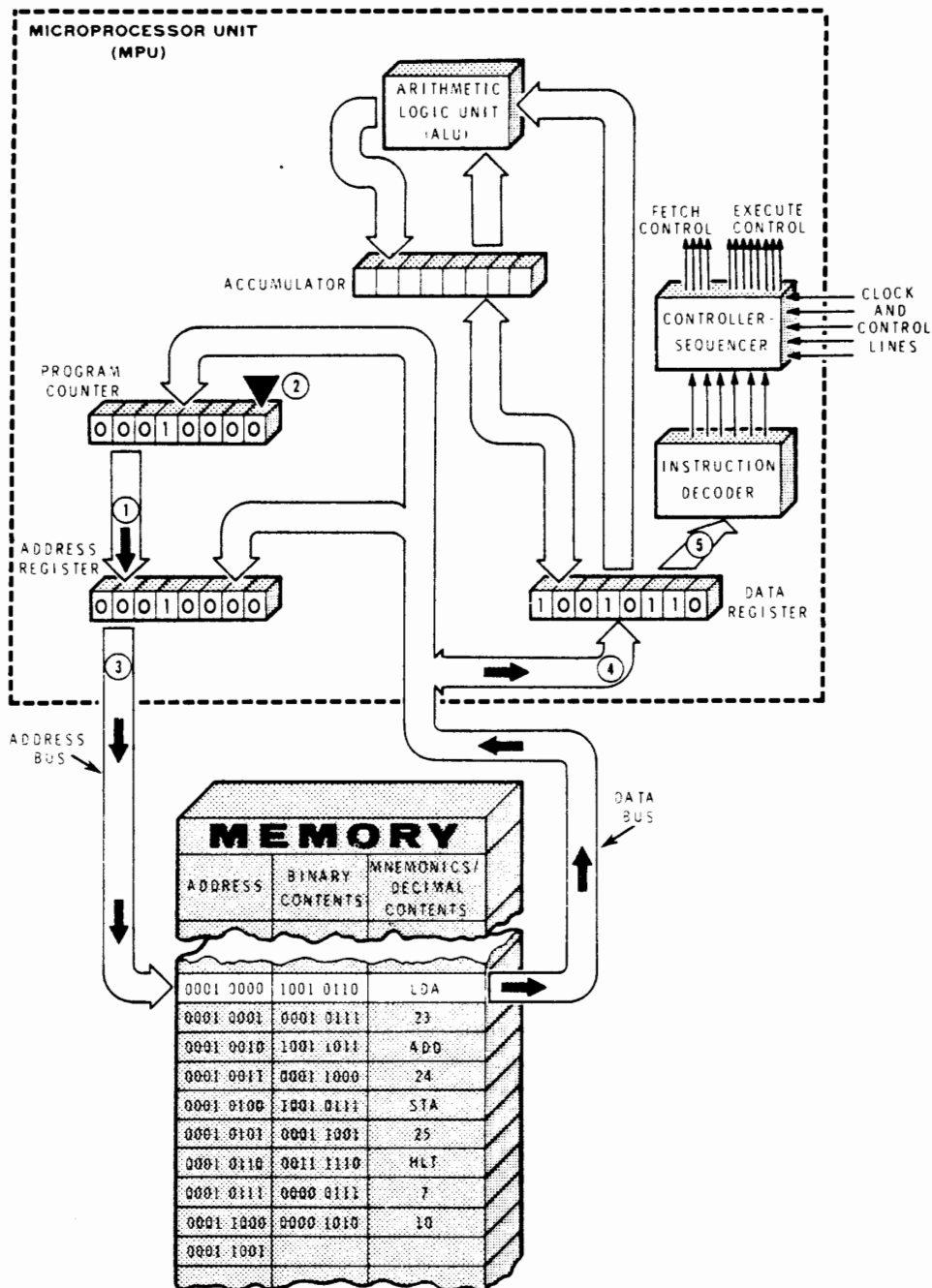
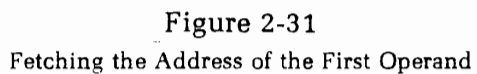


Figure 2-30  
Fetching the Opcode of the First In-  
struction.

The execute phase has two parts when direct addressing is indicated. Figure 2-31 illustrates the first half of the execute phase. During this half:

1. The contents of the program counter are transferred to the address register. This number is the memory location that holds the address of the operand.
2. The program counter is incremented to  $18_{10}$ .
3. The contents of the address register are placed on the address bus.
4. The contents of the selected memory location are placed on the data bus. However, in the direct addressing mode, this data is transferred to the address register. Thus,  $23_{10}$  goes into the address register, replacing the previous contents. After this cycle, the address register will appear as shown in Figure 2-32.





### Fetching the Address of the First Operand

During the second half of the execute phase, the operand is loaded into the accumulator as shown in Figure 2-32. The procedure is:

1. The address of the operand which is in the address register is placed on the address bus.
2. The operand is read out of memory location  $23_{10}$  and is transferred via the data bus to the data register.
3. The operand is transferred from the data register to the accumulator.

This completes the execution of the first instruction. Notice that the first operand ( $7_{10}$ ) is now in the accumulator.

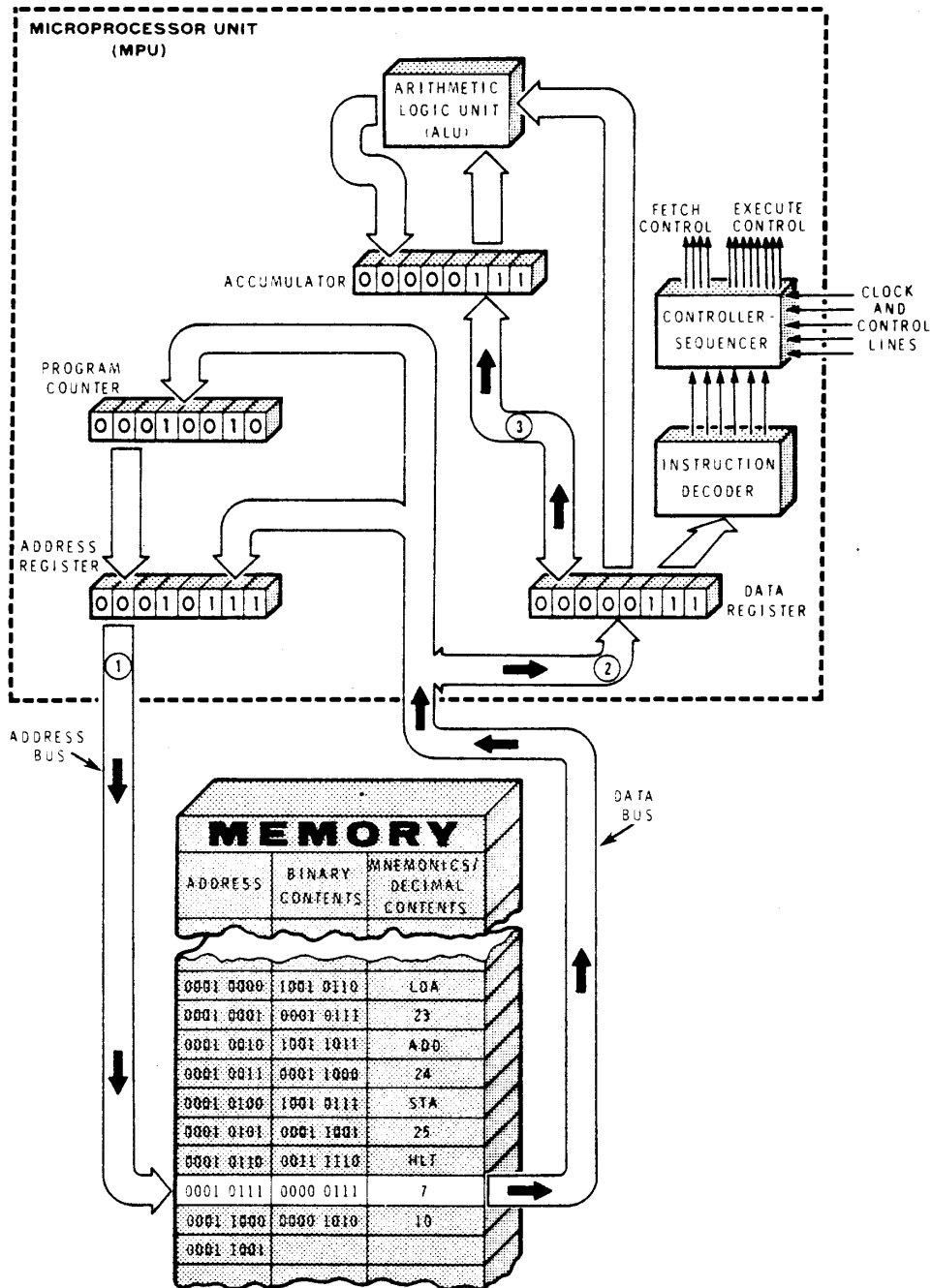


Figure 2-32  
Fetching the First Operand.

The fetch phase for the second instruction is similar to that of the first. As shown in Figure 2-33, it causes the opcode of the ADD instruction to be read out of address 18<sub>10</sub>. The opcode is transferred to the instruction decoder via the data bus and data register. In the process, the program counter is incremented to 19<sub>10</sub>.

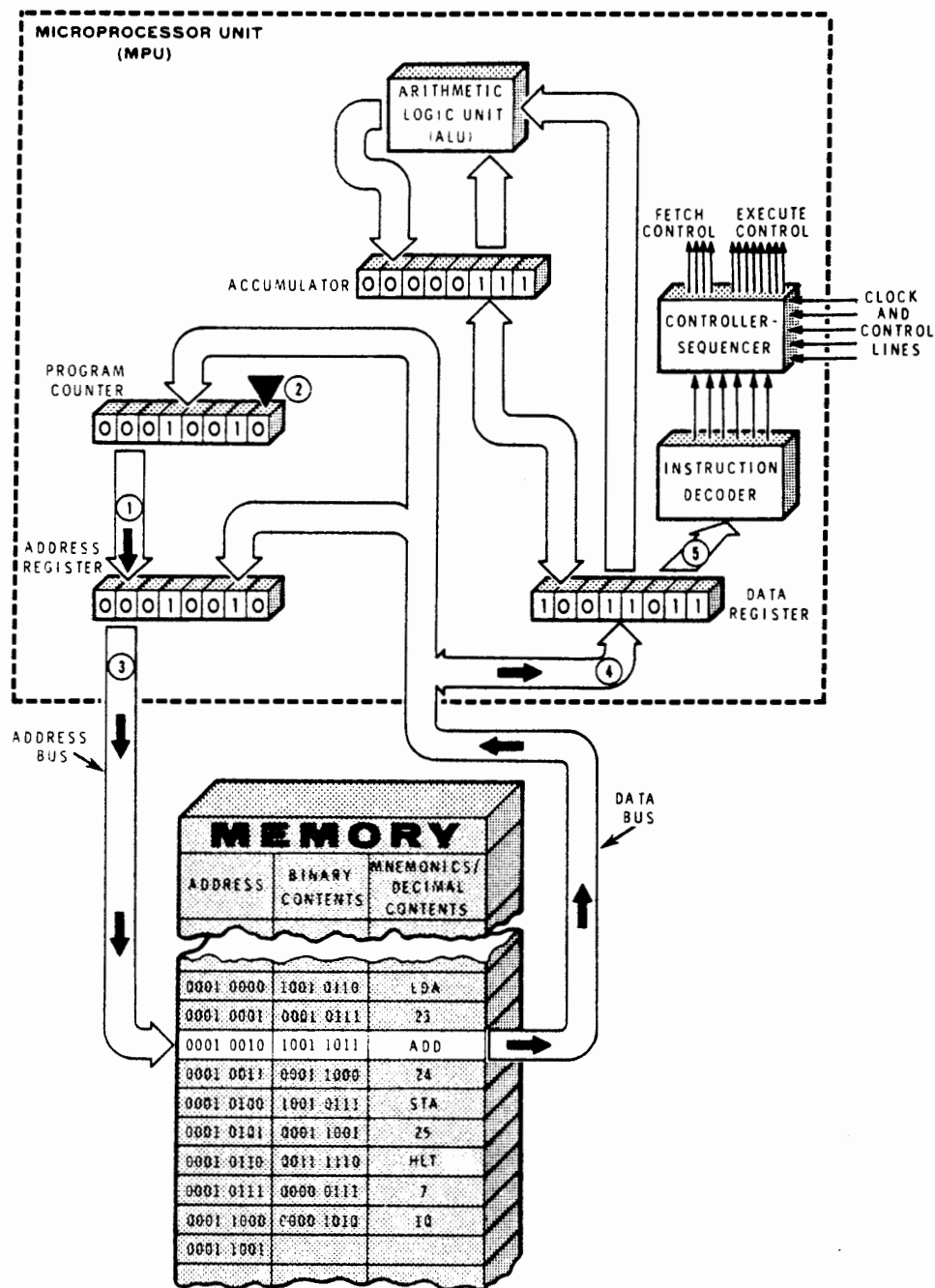


Figure 2-33

Fetching the Opcode of the Second Instruction.

The first half of the execute phase is illustrated in Figure 2-34. Here, the address of the second operand is read out of memory location 19<sub>10</sub> and is placed in the address register.

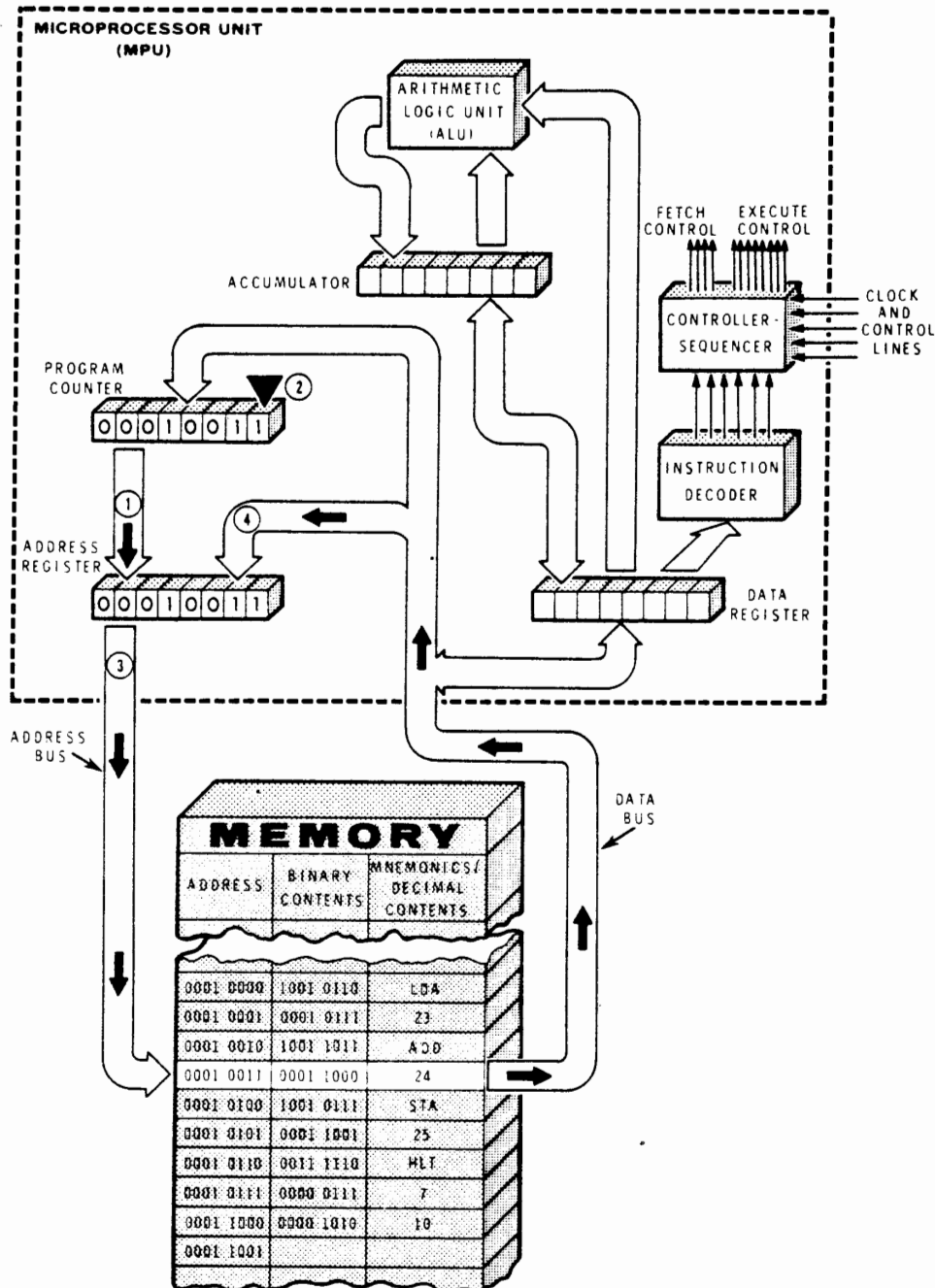


Figure 2-34  
Fetching the Address of the Second Operand.

Figure 2-35 illustrates the second cycle of the execute phase. Here the address of the second operand is transferred from the address register to the address bus. The address is  $24_{10}$ . Therefore, the contents of location  $24_{10}$  are placed on the data bus and transferred to the data register. That is, the second operand  $10_{10}$  is loaded into the data register. Then, the operand from the data register is made available at one input to the ALU. Simultaneously, the first operand which has been waiting in the accumulator is made available at the other input to the ALU. The ALU adds the two operands together, producing a result of  $17_{10}$ . This sum is put back in the accumulator, replacing the previous number.

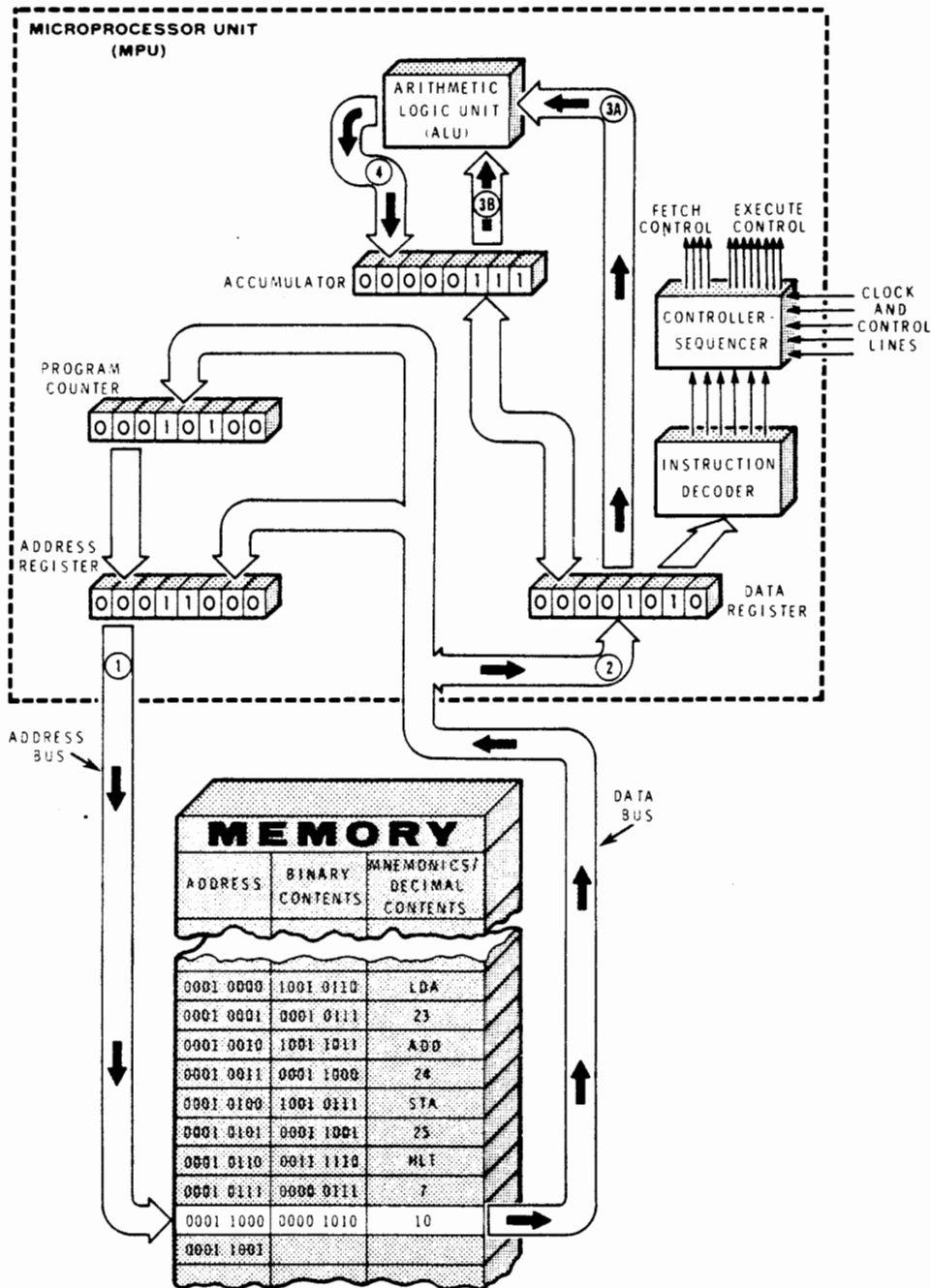


Figure 2-35  
Adding the Two Operands.

Now all that remains is to place the sum in memory. This is done by the STA 25<sub>10</sub> instruction. Since this is the next instruction in sequence, it will be fetched and executed next. The fetch phase is illustrated in Figure 2-36. It ends with the STA opcode being decoded.

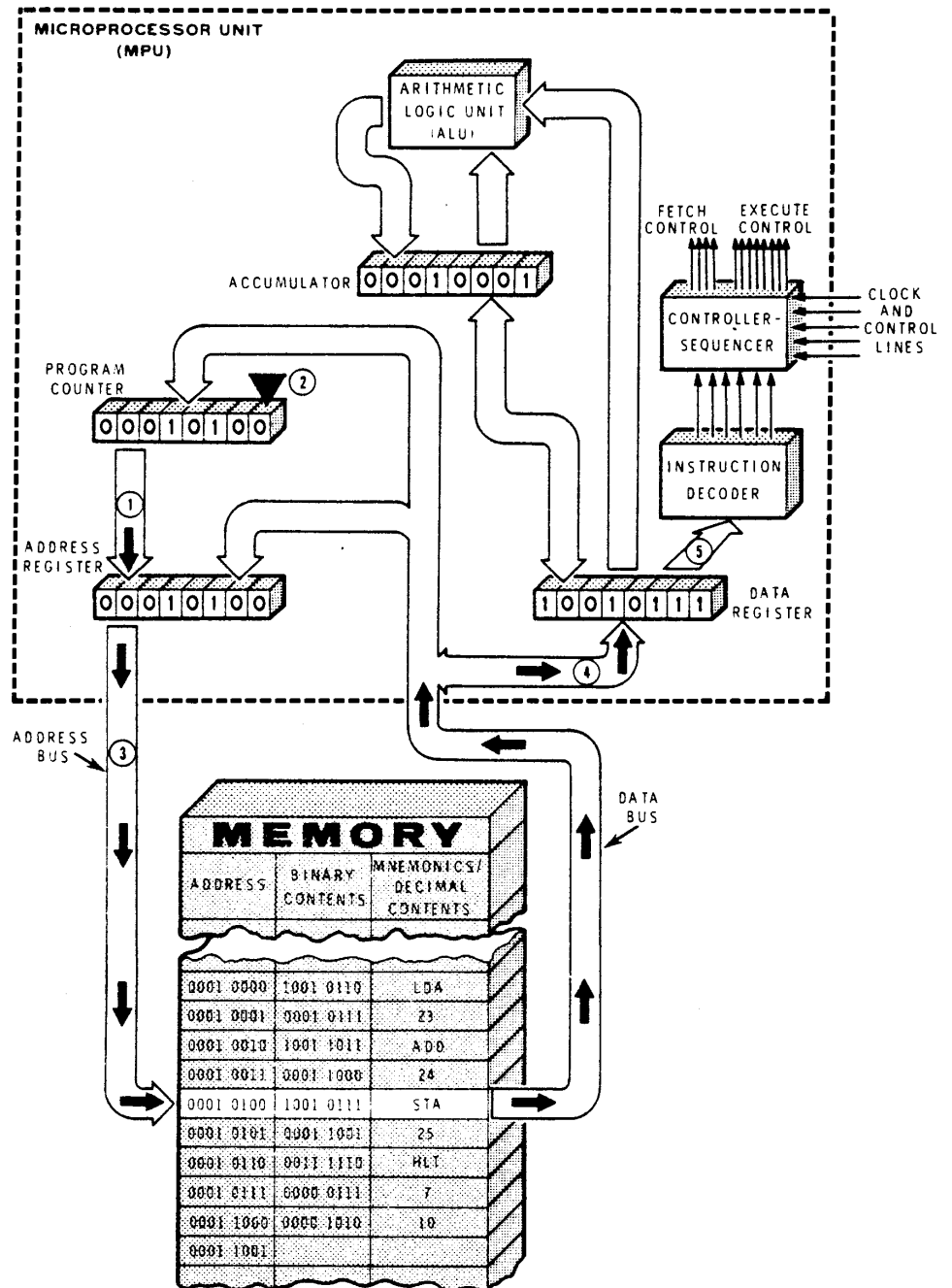


Figure 2-36  
Fetching the Third Opcode.



The first half of the execution phase of the STA instruction involves loading the address of the storage location into the address register. Figure 2-37 illustrates that this four-step procedure is identical to that performed for the previous two instructions. It ends with the address 25<sub>10</sub> in the address register.

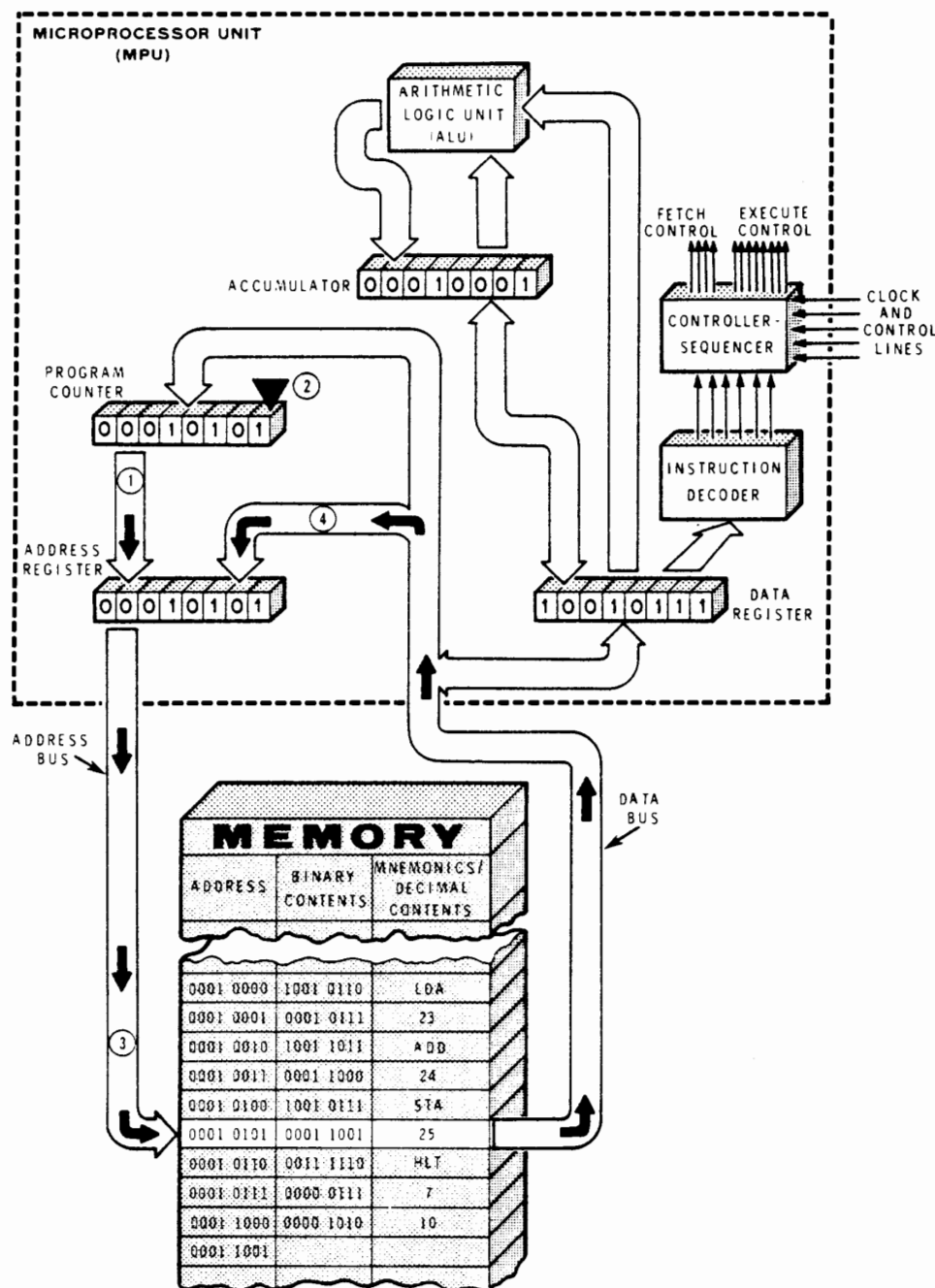


Figure 2-37  
 Fetching the Third Address.

During the final half of the execute phase, the contents of the accumulator are transferred to the data register and are then stored in the selected memory location. We have not yet discussed this operation in detail. Therefore, the step-by-step procedure is presented below. Refer to Figure 2-38 for the following steps:

1. The contents of the accumulator ( $17_{10}$ ) are transferred to the data register. At this point, the number  $17_{10}$  exists in both the accumulator and the data register.
2. The address at which this data is to be stored is placed on the address bus.
3. The contents of the data register are placed on the data bus.
4. The number on the data bus is written into the selected memory location. That is,  $17_{10}$  is written into memory location  $25_{10}$ .

Notice that, after this operation, the number  $17_{10}$  appears at memory location  $25_{10}$ , but it also appears in the accumulator. Thus, the number is merely "copied" into memory. It is also important to note that the previous contents of memory location  $25_{10}$  are lost whenever you write new data into this location. For this reason, you must be certain that you do not write into a location that contains an instruction or some byte of data that you will need later.

The program has now accomplished its goal. It has added 10 to 7 and has stored the sum back in memory. The last step in the program is the HLT instruction. The MPU fetches and executes this instruction next. The fetch and execute sequence for this instruction were discussed earlier and need not be repeated here.

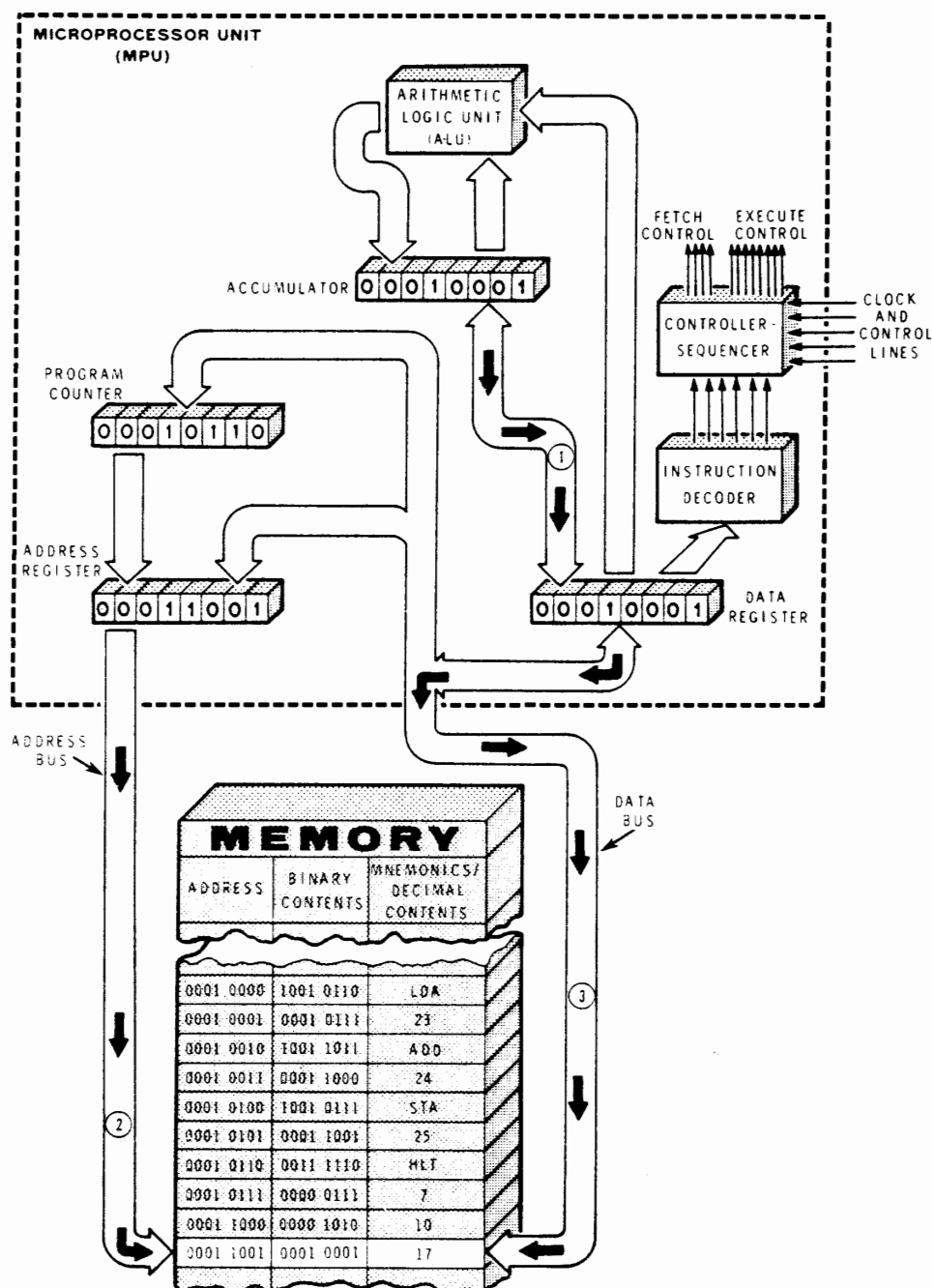


Figure 2-38  
Storing the Sum.

## Combining Addressing Modes

When writing programs, you can use the addressing mode that best suits your application. For example, the program that was just discussed can be shortened by using the immediate addressing mode with the first two instructions. Figure 2-39 compares two programs that do the same job.

Using direct addressing only, the program required ten bytes of memory. Its execution requires eleven MPU cycles. If you use immediate addressing for the first two instructions, the program requires eight bytes of memory. Furthermore, it can be executed in nine MPU cycles. Everything else being equal, the second approach would probably be preferred.

**A. USING DIRECT ADDRESSING**

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
00	96		Load accumulator direct with operand 1
01	07		which is stored at this address.
02	9B		Add to accumulator direct with operand 2
03	08		which is stored at this address.
04	97		Store the sum
05	09		at this address.
06	3E		Stop
07	21		Operand 1
08	17		Operand 2
09	—		Reserved for sum.

**B. COMBINING ADDRESSING MODES**

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
00	86		Load accumulator immediately with
01	21		Operand 1.
02	8B		Add to accumulator immediately with
03	17		Operand 2.
04	97		Store the sum
05	07		at this address.
06	3E		Stop
07	—		Reserved for sum.

Figure 2-39

By Combining the Addressing Modes,  
 We Can Save Memory Space And  
 Computer Time.

## Self-Test Review

41. What addressing mode is used by single byte instructions?
42. In the immediate addressing mode, what is the second byte of the instruction?
43. In the direct addressing mode, what is the second byte of the instruction?
44. In all addressing modes, what is the first byte of the instruction?
45. Define MPU cycle.
46. Which of the three addressing modes discussed so far requires the longest execution time?
47. Refer to Figure 2-39A. What number is loaded into the accumulator by the first instruction?
48. What number is added to the accumulator by the second instruction?
49. When the computer halts, what number will be in memory location 09?
50. Refer to Figure 2-39B. When the computer halts, what number will be in memory location 07?

## Answers

41. Inherent or implied.
42. The operand.
43. The address of the operand.
44. The opcode.
45. An MPU cycle is the time required to fetch a byte from memory.
46. The direct addressing mode.
47.  $21_{16}$  or  $33_{10}$ .
48.  $17_{16}$  or  $23_{10}$ .
49.  $38_{16}$  or  $56_{10}$ .
50.  $38_{16}$  or  $56_{10}$ .

A  
B  
C  
D  
A  
D  
C  
B  
C

~~A  
B  
C  
D  
A  
D  
C  
B  
C~~

### EXPERIMENT 3

Perform Experiment 3 in Unit 9 of this course. After you finish the experiment, return to this unit and complete the final examination.



## UNIT EXAMINATION

1. In microprocessor terminology, the number or piece of data that is operated upon is called the:
  - A. Operand.
  - B. Opcode.
  - C. Address.
  - D. Instruction.
2. The part of the instruction that tells the microprocessor what operation to perform is called the:
  - A. Operand.
  - B. Opcode.
  - C. Address.
  - D. Mnemonic.
3. The portion of the microcomputer in which instructions and data are stored is called the:
  - A. ALU.
  - B. MPU.
  - C. RAM.
  - D. Data bus.
4. An 8-bit byte in memory can represent an:
  - A. Opcode.
  - B. Operand.
  - C. Address.
  - D. All of the above.
5. During the fetch phase:
  - A. The opcode is fetched from memory and is decoded.
  - B. The address of the operand is fetched from memory and is decoded.
  - C. The operand is fetched from memory and is operated upon.
  - D. The program count is fetched from memory.

6. In what register is the result of an arithmetic operation normally placed?
  - A. The data register.
  - B. The address register.
  - C. The arithmetic logic unit (ALU).
  - D. The accumulator.
7. During the fetch and execute phases of the "load accumulator direct" instruction, the information on the data bus will be:
  - A. The operand address followed by the operand.
  - B. The program count, followed by the opcode, followed by the operand address, followed by the operand.
  - C. The opcode, followed by the operand address, followed by the operand.
  - D. The opcode, followed by the operand.
8. In the immediate addressing mode, the second byte of the instruction is the:
  - A. Opcode of the instruction.
  - B. Number that is to be operated upon.
  - C. Address of the operand.
  - D. Address of the opcode.
9. In the direct addressing mode, the second byte of the instruction is the:
  - A. Opcode of the instruction.
  - B. Number that is to be operated upon.
  - C. Address of the operand.
  - D. Address of the opcode.
10. Which of the following is normally a one-byte instruction?
  - A. Halt.
  - B. Add immediate.
  - C. Load accumulator direct.
  - D. Store accumulator direct.

11. At the start of the fetch phase, the program counter contains:

- A. The address of the operand to be fetched.
- B. The address of the opcode to be fetched.
- C. The opcode of the instruction.
- D. The operand.

12. Which register holds the opcode while it is being decoded?

- A. The address register.
- B. The accumulator.
- C. The data register.
- D. The program counter.

13. The program shown in Figure 2-40:

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS
00	86	LDA
01	00	00 <sub>16</sub>
02	97	STA
03	09	09 <sub>16</sub>
04	97	STA
05	0A	0A <sub>16</sub>
06	97	STA
07	0B	0B <sub>16</sub>
08	3E	HLT
09	—	—
0A	—	—
0B	—	—

Figure 2-40  
 Program for Question 13.

- A. Adds the contents of memory location 09, 0A, and 0B.
- B. Stores 00 in locations 09, 0A, 0B.
- C. Stores 09 in location 03, 0A in location 05, and 0B in location 07.
- D. Stores 0B in the accumulator.

14. The program shown in Figure 2-41:

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS
00	96	LDA
01	09	09 <sub>16</sub>
02	9B	ADD
03	09	09 <sub>16</sub>
04	9B	ADD
05	09	09 <sub>16</sub>
06	9B	ADD
07	09	09 <sub>16</sub>
08	3E	HLT
09	04	04 <sub>16</sub>

Figure 2-41

Program for Question 14.

- A. Multiplies 4 times 4 and holds the product in the accumulator.
- B. Multiplies 9 times 3 and holds the product in the accumulator.
- C. Multiplies 4 times 3 and stores the product in the accumulator.
- D. Multiplies 9 times 4 and holds the product in the accumulator.

15. The program shown in Figure 2-42:
- A. Swaps the contents of memory location 0D and 0E.
  - B. Stores  $AA_{16}$  in locations 0D, 0E, and 0F.
  - C. Stores  $BB_{16}$  in locations 0D, 0E, and 0F.
  - D. Adds AA and BB, storing the sum at location 0F.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS
00	96	LDA
01	0D	$0D_{16}$
02	97	STA
03	0F	$0F_{16}$
04	96	LDA
05	0E	$0E_{16}$
06	97	STA
07	0D	$0D_{16}$
08	96	LDA
09	0F	$0F_{16}$
0A	97	STA
0B	0E	$0E_{16}$
0C	3E	HLT
0D	AA	$AA_{16}$
0E	BB	$BB_{16}$
0F	—	—

Figure 2-42  
Program for Question 15.





# Individual Learning Program

## MICROPROCESSORS

### *UNIT 3* COMPUTER ARITHMETIC EE-3401

HEATH COMPANY  
BENTON HARBOR, MICHIGAN 49022

Copyright © 1977  
Heath Company  
All Rights Reserved  
Printed in the United States of America

## CONTENTS

Introduction .....	3-3
Unit Objectives .....	3-4
Unit Activity Guide .....	3-5
Binary Arithmetic .....	3-6
Two's Complement Arithmetic .....	3-26
Boolean Operations .....	3-35
Experiment 4 .....	3-44
Unit Examination .....	3-45
Examination Answers .....	3-47



## Unit 3

# COMPUTER ARITHMETIC

## INTRODUCTION

In this Unit you will complete your study of the binary number system. Since microprocessors use binary numbers for data and control, it is important that you become familiar with them.

Computer arithmetic involves many forms of number manipulation. In the pages that follow you will be given the fundamentals of binary mathematics: addition, subtraction, multiplication, and division. Then you will learn to perform two's complement arithmetic using binary numbers. Finally, you will be shown how the microprocessor performs the four basic Boolean logic operations. These logical operations include AND, OR, exclusive OR, and invert.

## UNIT OBJECTIVES

When you complete this Unit you will be able to:

1. Add two binary numbers.
2. Subtract one binary number from another.
3. Multiply one binary number by another.
4. Divide one binary number by another.
5. Derive the one's complement of a binary number.
6. Derive the two's complement of a binary number.
7. Add binary numbers using two's complement arithmetic.
8. Manipulate binary numbers using the AND operation.
9. Manipulate binary numbers using the OR operation.
10. Manipulate binary numbers using the exclusive OR operation.
11. Logically invert binary numbers.

## UNIT ACTIVITY GUIDE

	Completion Time
<input type="checkbox"/> Read section on Binary Arithmetic.	_____
<input type="checkbox"/> Answer Self-Test Review Questions 1-11.	_____
<input type="checkbox"/> Read section on Two's Complement Arithmetic.	_____
<input type="checkbox"/> Answer Self-Test Review Questions 12-21.	_____
<input type="checkbox"/> Read section on Boolean Operations.	_____
<input type="checkbox"/> Answer Self-Test Review Questions 22-30.	_____
<input type="checkbox"/> Perform Experiment 4.	_____
<input type="checkbox"/> Complete Unit Examination.	_____
<input type="checkbox"/> Review Examination Answers.	_____

## BINARY ARITHMETIC

A number system can be used to perform two basic operations: addition and subtraction. But by using addition and subtraction, you can then perform multiplication, division, and any other numerical operation. In this section, binary arithmetic (addition, subtraction, multiplication, and division) will be examined, using decimal arithmetic as a guide.

### Binary Addition

Binary addition is performed somewhat like decimal addition. If two decimal numbers, 56719 and 31863, are added together, the sum 88582 is obtained. You could analyze the details of this operation in the following manner.

*NOTE: In the following explanations, the term "first column" refers to the first column of figures you work with in the problem — the column on the right (9, 3, and 2 in the following example). The term "second column" refers to the second column you work with, etc.*

Carry:	00101
Addend:	56719
Augend:	+ 31863
Sum:	<u>88582</u>

Adding the first column, decimal numbers 9 and 3, gives the sum of 12. This is expressed in the sum as the digit 2 with a carry of 1. The carry is then added to the next column. Adding the second column decimal numbers 1 and 6, and the carry from the first column, gives the sum of 8, with no carry. This process continues until all of the columns (including carries) have been added. The sum represents the numeric value of the addend and augend. (The **addend** is the number to be added to another number, while the **augend** is the number to which the addend is added.)

When you add two binary numbers, you perform the same operation. Figure 3-1 summarizes the four rules of addition with binary numbers.

1.  $0 + 0 = 0$
2.  $0 + 1 = 1$
3.  $1 + 1 = 0$  with a carry of 1.
4.  $1 + 1 + 1 = 1$  with a carry of 1.

Figure 3-1  
Rules for binary addition.

To illustrate the process of binary addition, let's add 1101 to 1101.

Carry:	1101
Addend:	1101
Augend:	+ 1101
Sum:	<u>11010</u>

In the first column, 1 plus 1 equals 0 with a carry of 1 to the second column. This agrees with rule 3. In the second column, 0 plus 0 equals 0 with no carry. To this sum, the carry from the first column is added. Thus, 0 plus 1 equals 1 with no carry. These two additions in the second column give a total sum of 1 with a carry of 0. Rules 1 and 2 were used to obtain the sum.

In column three, 1 plus 1 equals 0 with a carry of 1. To this sum, the second column carry is added. This yields a third column sum of 0 with a carry of 1 to column four. Rules 3 and 1 were used to obtain the sum.

In column four, 1 plus 1 equals 0 with a carry of 1. To this sum, the third column carry is added. This yields a fourth column sum of 1 with a carry to the fifth column. Rule 4 allows you to add three binary 1's and obtain 1 with a carry of 1.

In column five, there is no addend or augend. Therefore, you can assume rule 2 and add the carry to 0 to obtain the sum of 1. Thus, the sum of  $1101_2$  plus  $1101_2$  equals  $11010_2$ . You can verify this by converting the binary numbers to decimal numbers.

Now study the following two examples of binary addition, where  $10001111_2$  is added to  $10110101_2$  and  $111011_2$  is added to  $11001100_2$ .

Carry:	10111111
Addend:	10110101
Augend:	+ 10001111
Sum:	<u>101000100</u>

Carry:	11111000
Addend:	11001100
Augend:	+ 00111011
Sum:	<u>100000111</u>

When binary addition is performed with a microprocessor, 8-bit numbers are generally used. As shown in the last example, two zeros were added after the MSB of the augend to produce an 8-bit number. After addition, a 1 in the ninth bit is represented as the "carry" bit by the microprocessor. This will be explained in a later unit.

## Binary Subtraction

Binary subtraction is performed exactly like decimal subtraction. Therefore, before binary subtraction can be attempted, decimal subtraction should be reexamined. You know that if decimal 5486 is subtracted from 8303, the difference 2817 is obtained.

Minuend after borrow:	7 12 9 13
Minuend:	8 3 0 3
Subtrahend:	<u>-5 4 8 6</u>
Difference:	2 8 1 7

Because the digit 6 in the subtrahend is larger than the digit 3 in the minuend, a 1 is borrowed from the next higher-order digit in the minuend. If that digit is 0, as in this example, 1 is borrowed from the next higher-order digit that contains a number other than 0. That digit is reduced by 1 (from 3 to 2 in this example) and the digits skipped in the minuend are given the value 9. This is equivalent to removing 1 from 30 with the result of 29, as in this example. In the decimal system, the digit borrowed has the value of ten. Therefore, the minuend digit now has the value 13, and 6 from 13 equals 7.

In the second column, 8 from 9 equals 1. Since the subtrahend is larger than the minuend in the third column, 1 is borrowed from the next higher-order digit. This raises the minuend value from 2 to 12, and 4 from 12 equals 8. In the fourth column, the minuend was reduced from 8 to 7 because of the previous borrow, and 5 from 7 equals 2.

Whenever 1 is borrowed from a higher-order digit, the borrow is equal in value to the radix or base of the number system. Therefore, a borrow in the decimal number system equals ten, while a borrow in the binary number system equals two.

When you subtract one binary number from another, you use the same method described for decimal subtraction. Figure 3-2 summarizes the four rules for binary subtraction.

1.  $0 - 0 = 0$
2.  $1 - 1 = 0$
3.  $1 - 0 = 1$
4.  $0 - 1 = 1$  with a borrow of 1.

Figure 3-2  
Rules for binary subtraction.

To illustrate the process of binary subtraction, let's subtract 1101 from 11011.

Minuend after borrow:	0 10 10 1 1
Minuend:	1 1 0 1 1
Subtrahend:	<u>1 1 0 1</u>
Difference:	1 1 1 0

The "minuend after borrow" now shows the value of each minuend digit after a borrow occurs. Remember that binary 10 equals decimal 2.

In the first column, 1 from 1 equals 0 (rule 2). Then, 0 from 1 in the second column equals 1 (rule 3). In the third column, 1 from 0 requires a borrow from the fourth column. Thus, 1 from 10<sub>2</sub> equals 1 (rule 4). The minuend in the fourth column is now 0, from the previous borrow. Therefore, a borrow is required from the fifth column, so that 1 from 10<sub>2</sub> in the fourth column equals 1 (rule 4). Because of the previous borrow, the minuend in

the fifth column is now 0 and the subtrahend is 0 (nonexistent), so that 0 from 0 equals 0 (rule 1). The 0 in the fifth column is not shown in the difference because it is not a significant bit. Thus, the difference between  $11011_2$  and  $1101_2$  is  $1110_2$ . You can verify this by converting the binary numbers to decimal numbers.

As a further example of binary subtraction, subtract  $00100101_2$  from  $11000100_2$ , as shown below. Then proceed to the next example and subtract  $10111010_2$  from  $11101110_2$ .

Minuend after borrow:	1 0 1 1 1 10 1 10
Minuend:	1 1 0 0 0 1 0 0
Subtrahend:	<u>-0 0 1 0 0 1 0 1</u>
Difference:	1 0 0 1 1 1 1 1

Minuend after borrow:	0 0 10 10 1 1 1 0
Minuend:	1 1 1 0 1 1 1 0
Subtrahend:	<u>-1 0 1 1 1 0 1 0</u>
Difference:	0 0 1 1 0 1 0 0

When a borrow is required in the minuend, 1 is obtained from the next high-order bit that contains a 1. That bit then becomes 0, and all bits skipped (0 value bits) are given the value 1. This is equivalent to removing 1 from  $1000_2$  with the result of  $0111_2$ .

As with binary addition, microprocessors generally perform subtraction on 8-bit number groups. In the previous example, the answer contained only six significant bits, but two 0 bits were added to maintain the 8-bit grouping. This would also be true for the minuend and subtrahend.

Subtraction of a large number from a smaller number will be described in a later section of this Unit.



## Binary Multiplication

Multiplication is a short method of adding a number to itself as many times as it is specified by the multiplier. However, if you were to multiply  $324_{10}$  by  $223_{10}$ , you would probably use the following method.

Multiplicand:	324
Multiplier:	$\times 223$
First partial product:	<u>972</u>
Second partial product:	648
Third partial product:	<u>648</u>
Carry:	<u>0121</u>
Final product:	72252

Using this short form of multiplication, you multiply the multiplicand by each digit of the multiplier and then sum the partial products to obtain the final product. Note that, for convenience, the additive carries are set-down under the partial products rather than over them as in normal addition.

Binary multiplication follows the same general principles as decimal multiplication. However, with only two possible multiplier bits (1 or 0), binary multiplication is a much simpler process. Figure 3-3 lists the rules of binary multiplication. These rules are used to multiply  $1111_2$  by  $1101_2$  on the next page.

1.  $0 \times 0 = 0$

2.  $0 \times 1 = 0$

3.  $1 \times 0 = 0$

4.  $1 \times 1 = 1$

Figure 3-3

Rules for binary multiplication.

Multiplicand:	1111
Multiplier:	<u>×1101</u>
First partial product:	1111
Second partial product:	<u>0000</u>
Carry:	<u>0000</u>
Sum of partial products:	1111
Third partial product:	<u>1111</u>
Carry:	<u>111100</u>
Sum of partial products:	1001011
Fourth partial product:	<u>1111</u>
Carry:	<u>1111000</u>
Final product:	11000011

As with decimal multiplication, you multiply the multiplicand by each bit in the multiplier and add the partial sums. First you multiply  $1111_2$  by the least significant multiplier bit (1) and set down the partial product so the least significant bit (LSB) is under the multiplier bit. Then you multiply the multiplicand by the next multiplier bit (0) and set down the partial product so the LSB is under the multiplier bit. Now that there are two partial products, they should be added. Although it is possible to add more than two binary numbers, keeping track of the multiple carries may become confusing. Therefore, for these examples, add only two partial products at a time.

Notice that the first partial product is identical to the multiplicand. The second partial product is all zeros. Since the binary number system contains only ones and zeros, the partial product will always equal either the multiplicand or zero. Because of this, you can obtain the third partial product by copying the multiplicand. Begin with the LSB under the third multiplier bit. Add this value to the previous partial sum. Now obtain the fourth partial product by copying the multiplicand. Begin with the LSB under the fourth multiplier bit. Add this value to the previous partial sum. This is the final product. You can verify the result by converting the binary numbers to decimal.

Reexamine the illustration for the previous multiplication example. Notice that binary multiplication is a process of shift and add. For each 1 bit in the multiplier you copy down the multiplicand, beginning with the LSB under the bit. You can ignore any zeros in the multiplier. But do not make the mistake of setting down the multiplicand under the 0 bit.

To make sure you fully understand binary multiplication, multiply  $1001_2$  by  $1100_2$  and then multiply  $1101_2$  by  $1111_2$ .

Multiplicand:	1001
Multiplier:	$\times 1100$
First partial product:	<u>0000</u>
Second partial product:	<u>0000</u>
Carry:	<u>0000</u>
Sum of partial products:	00000
Third partial product:	<u>1001</u>
Carry:	<u>00000</u>
Sum of partial products:	100100
Fourth partial product:	<u>1001</u>
Carry:	<u>000000</u>
Final product:	1101100

Multiplicand:	1101
Multiplier:	$\times 1111$
First partial product:	<u>1101</u>
Second partial product:	<u>1101</u>
Carry:	<u>11000</u>
Sum of partial products:	100111
Third partial product:	<u>1101</u>
Carry:	<u>100100</u>
Sum of partial products:	1011011
Fourth partial product:	<u>1101</u>
Carry:	<u>1111000</u>
Final product:	11000011

In the first of these last two examples, the two zeros in the multiplier were included in the multiplication process. This was to insure that the multiplicand was copied down under the proper multiplier bits. The multiplication process could have been represented in this manner:

Multiplicand:	1001
Multiplier:	$\times 1100$
Third partial product:	<u>100100</u>
Fourth partial product:	<u>1001</u>
Carry:	<u>000000</u>
Final product:	1101100

Remember, just as in decimal multiplication, you must keep track of any zeros by setting a zero in the product under the 0 bit in the multiplier. This is very important when the zero occupies the LSB.

## Binary Division

Division is the reverse of multiplication. Therefore, it is a procedure for determining how many times one number can be subtracted from another. The process you are probably familiar with is called "long" division. If you were to divide decimal 181 by 45, you would obtain the quotient,  $4\frac{1}{45}$ , as follows:

		004	Quotient
Divisor	45	$\overline{)181}$	Dividend
		180	
		$\underline{1}$	Remainder

Using long division, you would examine the most significant digit in the dividend and determine if the divisor was smaller in value. In this example the divisor is larger, so the quotient is zero. Next, you examine the two most significant digits. Again the divisor is larger, so the quotient is again zero. Finally, you examine the whole dividend and discover it is approximately four times the divisor in value. Therefore, you give the quotient a value of 4. Next, you subtract the product of 45 and 4 (180) from the dividend. The difference of one represents a fraction of the divisor. This fraction is added to the quotient to produce the correct answer of  $4\frac{1}{45}$ .

Binary division is performed in a similar manner. However, binary division is a simpler process since the number base is two rather than ten. First, let's divide  $100011_2$  by  $101_2$ .

		000111	Quotient
Divisor:	101	$\overline{)100011}$	Dividend
		101	
		$\underline{111}$	Remainder
		101	
		$\underline{101}$	Remainder
		101	
		$\underline{0}$	Remainder

Using long division, you examine the dividend beginning with the MSB and determine the number of bits required to exceed the value of the divisor. When you find this value, place a one in the quotient and subtract the divisor from the selected dividend value. Then carry the next least significant bit in the dividend down to the remainder. If you can subtract the divisor from the new remainder, place a one in the quotient. Then subtract the divisor from the remainder and carry the next least signifi-

cant bit in the dividend (LSB in this example) down to the remainder. If the divisor can be subtracted from the new remainder, place a one in the quotient and subtract the divisor from the remainder. Continue the process until all of the dividend bits have been carried down. Then express any remainder as a fraction of the divisor in the quotient. Thus,  $100011_2$  divided by  $101_2$  equals  $111_2$ . You can verify the answer by converting the binary numbers to decimal.

To make sure you fully understand binary division, work out the following examples of long division. Divide  $101000_2$  by  $1000_2$  and then divide  $100111_2$  by  $110_2$ .

		000101	Quotient
Divisor	1000	) 101000	Dividend
		1000 ↓ ↓	
		1000	Remainder
		1000	
		0	Remainder

		000110.1	Quotient
Divisor:	110	) 100111.0	Dividend
		110 ↓ ↓	
		111	Remainder
		110 ↓ ↓	
		11 0	Remainder
		11 0	
		0	Remainder

In the second example, the quotient was not a whole number, but rather a whole number plus a fraction (remainder divided by the divisor). The answer  $110-11/110$  is correct. You could have left the answer in this form or, as in the example, continue the division process until the remainder was zero. This is made possible by adding a sufficient number of zeros after the binary point to permit division by the divisor. In the previous example, only one zero was added after the binary point. As you learned in Unit 1, adding zeros after the binary point will not affect the value of the number. Note that some numbers cannot be solved in this manner (e.g., decimal  $1/3$ ).

## Representing Negative Numbers

Until now, we have been examining binary arithmetic using unsigned numbers. However, when you perform some arithmetic operations with a microprocessor, you must be able to express both positive and negative (signed) numbers. Over the years three methods have been developed for representing signed numbers. Of these, only one method has survived. The two older methods will be examined first, and then the system that is used today.

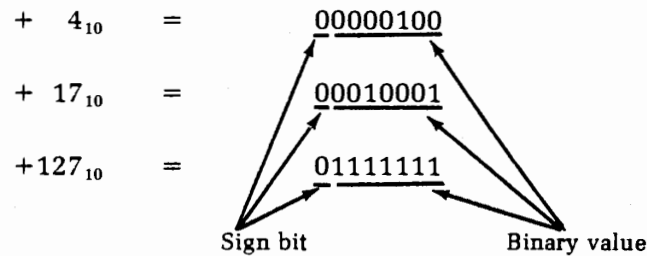
**SIGN AND MAGNITUDE.** Using this system, a binary number contained both the sign (+ or -) and the value of the number. Therefore, positive and negative values were expressed as follows:

$$\begin{array}{rcl} +45_{10} & = & \underline{00101101}_2 \\ & & \swarrow \quad \searrow \\ & & \text{SIGN} \quad \text{MAGNITUDE} \\ & & \swarrow \quad \searrow \\ -45_{10} & = & \underline{10101101}_2 \end{array}$$

The MSB of the binary number indicated the sign, while the remaining bits contained the value of the number. As you can see, a zero sign bit indicated a positive value, while a one sign bit indicated a negative value.

While this method of representing negative numbers may seem logical, its popularity was short-lived. Because it required complex and slow arithmetic circuitry, it was abandoned long before microprocessors were invented.

**ONE'S COMPLEMENT.** Another method of representing negative numbers became popular in the early days of computers. It was called the one's complement method. Using this system, positive numbers were represented in the same way as in the sign-magnitude system. That is, the MSB in any number was considered to be a sign bit. A sign bit of 0 represented positive. Using 8-bit numbers, positive values were represented like this:



Negative numbers were represented by the **one's complement** of the positive value. The one's complement of a number is formed by changing all 0's to 1's and all 1's to 0's. As shown above,  $+4_{10}$  is represented as 0 0000100<sub>2</sub>. By changing all 0's to 1's and all 1's to 0's, the representation for  $-4_{10}$  was formed. In this case:

$$- 4_{10} = \underline{1} \underline{1111011}_2$$

Notice that all the bits, including the sign bit, were inverted. In the same way:

$$- 17_{10} = \underline{1} \underline{1101110}_2$$

$$- 127_{10} = \underline{1} \underline{0000000}_2$$

The one's complement method is not used for representing signed numbers in microprocessors. However, as you will see later, you may still be called upon to find the one's complement of a number. Remember, you do this by simply changing all 0's to 1's and all 1's to 0's.

Figure 3-4 shows an interesting relationship. In the first column, 8-bit patterns of 0's and 1's are shown. The second column shows the decimal number that each pattern represents if you consider the pattern to be an unsigned binary number. Notice that an 8-bit pattern can represent unsigned numbers between 0 and  $255_{10}$ .

The third column shows the decimal number that each pattern represents if you consider the pattern to be a one's complement binary number. Notice that the range of numbers is from  $-127_{10}$  to  $+127_{10}$ . Notice also that there are two representations of zero. The pattern  $0000\ 0000_2$  represents  $+0$  while its one's complement ( $1111\ 1111_2$ ) represents  $-0$ .

**TWO'S COMPLEMENT.** The method used to represent signed numbers in microprocessors is called two's complement. In this system, positive numbers are represented just as they were with the sign-and-magnitude method and the one's complement method. That is, it uses the same bit pattern for all positive values up to  $+127_{10}$ . However, negative numbers are represented as the two's complement of positive numbers.

The two's complement of a number is formed by taking the one's complement and then adding 1. For example if you work with 8-bit numbers and use the two's complement system,  $+4_{10}$  is represented by  $00000100_2$ . To find  $-4_{10}$ , you must take the two's complement of this number. You do this by first taking the one's complement, which is  $11111011_2$ . Next, add 1 to form the two's complement:

$$\begin{array}{r} 11111011_2 \\ + \quad 1 \\ \hline 11111100_2 \end{array}$$

Thus, the two's complement representation of  $-4_{10}$  is  $11111100_2$ .

To be sure you have the idea, look at a second example: how do you express  $-17_{10}$  as an 8-bit two's complement number? Start with the two's complement representation of  $+17_{10}$ , which is  $00010001_2$ . Take the one's complement by changing all 0's to 1's and 1's to 0's. Thus, the one's complement of  $+17_{10}$  is  $11101110_2$ . Next, find the two's complement by adding 1:

$$\begin{array}{r} 11101110_2 \\ + \quad 1 \\ \hline 11101111_2 \end{array}$$



BIT PATTERN	UNSIGNED BINARY	1's COMPLEMENT
00000000	0	+0
00000001	1	+1
00000010	2	+2
00000011	3	+3
.	.	.
.	.	.
.	.	.
.	.	.
01111100	124	+124
01111101	125	+125
01111110	126	+126
01111111	127	+127
10000000	128	-127
10000001	129	-126
10000010	130	-125
10000011	131	-124
.	.	.
.	.	.
.	.	.
.	.	.
11111100	252	-3
11111101	253	-2
11111110	254	-1
11111111	255	-0

Figure 3-4

Table of bit pattern values for unsigned binary numbers and 1's complement numbers.

Figure 3-5 compares unsigned, two's complement, and one's complement numbers. Several 8-bit patterns are shown on the left. The other three columns show the decimal number represented by these patterns.

Notice that the range of 8-bit two's complement numbers is from  $-128_{10}$  to  $+127_{10}$ . Notice also that there is only one representation for 0.

If this table included all  $256_{10}$  possible 8-bit patterns, you could look up any pattern to see what number it represents. The patterns which have 0 as their MSB are easy to determine without a table. The pattern represents the binary number directly. But what decimal number is represented by the two's complement number  $11110011$ ? You should know that this represents some negative number because the MSB is a 1.

Actually, you can determine the value very easily by simply taking the two's complement to find the equivalent positive number. Remember, you find the two's complement, by taking the one's complement and adding 1. The one's complement is  $00001100_2$ . Thus, the two's complement is:

$$\begin{array}{r}
 00001100_2 \\
 + \quad \quad 1 \\
 \hline
 00001101_2 \quad \text{or } +13_{10}
 \end{array}$$

Since the two's complement of  $11110011_2$  represents  $+13_{10}$ , then  $11110011_2$  must equal  $-13_{10}$ .

BIT PATTERN	UNSIGNED BINARY	2's COMPLEMENT	1's COMPLEMENT
00000000	0	0	+0
00000001	1	+1	+1
00000010	2	+2	+2
00000011	3	+3	+3
.	.	.	.
.	.	.	.
.	.	.	.
.	.	.	.
01111100	124	+124	+124
01111101	125	+125	+125
01111110	126	+126	+126
01111111	127	+127	+127
10000000	128	-128	-127
10000001	129	-127	-126
10000010	130	-126	-125
10000011	131	-125	-124
.	.	.	.
.	.	.	.
.	.	.	.
.	.	.	.
11111100	252	-4	-3
11111101	253	-3	-2
11111110	254	-2	-1
11111111	255	-1	-0

Figure 3-5

Table of bit pattern values for unsigned binary, 2's complement and 1's complement numbers.

## Self-Test Review

- \_\_\_\_\_ and \_\_\_\_\_ are the two basic operations that can be performed with a number system.
- Add the following binary numbers.
 

A.     10011011 +00010111 <u>          </u>	B.     11000110 +00110001 <u>          </u>	C.     10000110 +00110110 <u>          </u>
---	---	---
- Subtract the following binary numbers.
 

A.     11011011 -10110010 <u>          </u>	B.     10001011 -10000001 <u>          </u>	C.     11011001 -00111011 <u>          </u>
---	---	---
- Multiply the following binary numbers.
 

A.     1011 ×1101 <u>          </u>	B.     1101 ×1001 <u>          </u>	C.     1100 ×1100 <u>          </u>
---	---	---
- Solve for the quotient in the following groups.
 

A.     _____ 101 $\overline{)1001011}$	B.     _____ 11 $\overline{)111001}$	C.     _____ 1101 $\overline{)11110111}$
---	---	---
- $10001111_2$  represents decimal \_\_\_\_\_ in sign/magnitude notation.

7. The 1's complement of  $00010110_2$  is \_\_\_\_\_.
8. The 2's complement of  $00010110_2$  is \_\_\_\_\_.
9. The 2's complement number 11100110 represents the decimal number \_\_\_\_\_.
10. Find the signed decimal equivalents of the following two's complement numbers.

<u>Two's Complement Number</u>	<u>Decimal Number</u>
00000111	
10000111	
11111111	
01110000	
10000000	

11. Find the two's complement representation for the following signed decimal numbers.

<u>Decimal Number</u>	<u>Two's Complement Number</u>
+32	
-32	
+73	
- 7	
-120	

## Answers

1. Addition, subtraction.

2. A.      Carry:                      00011111  
              Addend:                    10011011  
              Augend:                    + 00010111  
              Sum:                        10110010

B.      11110111.

C.      10111100.

3. A.      Minuend after borrow:      1 0 10 1 1 0 1 1  
              Minuend:                    1 1 0 1 1 0 1 1  
              Subtrahend:                - 1 0 1 1 0 0 1 0  
              Difference:                   1 0 1 0 0 1

B.      1010.

C.      10011110.

4. A.      Multiplicand:                    1011  
              Multiplier:                    × 1101  
              First partial product:            1011  
              Second partial product:           00000  
              Carry:                        0000  
              Sum of partial products:           01011  
              Third partial product:            101100  
              Carry:                        01000  
              Sum of partial products:           110111  
              Fourth partial product:           1011000  
              Carry:                        1110000  
              Final product:                   10001111

B.      1110101.

C.      10010000

5.      A.      Divisor:    101     $\begin{array}{r} 0001111 \\ 1001011 \overline{) 1001011} \\ \underline{101} \\ 1000 \\ \underline{101} \\ 111 \\ \underline{101} \\ 101 \\ \underline{101} \\ 0 \end{array}$       Quotient  
Dividend  
Remainder  
Remainder  
Remainder  
Remainder

B.      10011.

C.      10011.

6.      -15.

7.       $11101001_2$ .

8.       $11101010_2$ .

9.      First, find the two's complement of 11100110 by changing 1's to 0's; 0's to 1's; and adding 1:

$\begin{array}{r} 00011001 \\ \underline{\phantom{000}1} \\ 00011010 \end{array}$

Since this number represents  $+26_{10}$ , the original number must have represented  $-26_{10}$ .

10.	<u>Two's Complement Number</u>	<u>Decimal Number</u>
	00000111	+7
	10000111	-121
	11111111	-1
	01110000	+112
	10000000	-128

11.	<u>Decimal Number</u>	<u>Two's Complement Number</u>
	+32	00100000
	-32	11100000
	+73	01001001
	-7	11111001
	-120	10001000

## TWO'S COMPLEMENT ARITHMETIC

In the previous section, you saw that signed numbers are represented in microprocessors in two's complement form. In this section you will see why.

In digital electronic devices such as computers, simple circuits cost less and operate faster than more complex ones. Two's complement numbers are used with arithmetic because they allow the simplest, cheapest, and fastest circuits.

A characteristic of the two's complement system is that both signed and unsigned numbers can be added by the same circuit. For example, suppose you wish to add the **unsigned** numbers  $132_{10}$  and  $14_{10}$ . The addition looks like this:

Addend:	$10000100_2$	$132_{10}$
Augend:	$00001110_2$	$+ 14_{10}$
Sum:	$10010010_2$	$146_{10}$

As you saw in the previous unit, the microprocessor has an ALU circuit that can add unsigned binary numbers in this way. The adder in the ALU is designed so that when the bit pattern  $10000100$  appears at one input and  $00001110$  appears at the other, the bit pattern  $10010010$  appears at the output.

The question arises, "How does the ALU know that the bit patterns at the inputs represent unsigned numbers and not two's complement numbers?" The answer is "it doesn't." The ALU always adds as if the inputs were unsigned binary numbers. Nevertheless, it still produces the correct sum even if the inputs are signed two's complement numbers.

Look at the example given above. If you assume that the inputs are two's complement signed numbers, then the addend, augend, and sum are:

Addend:	$10000100_2$	$-124_{10}$
Augend:	$00001110_2$	$+ 14_{10}$
Sum:	$10010010_2$	$-110_{10}$

Notice that the bit patterns are the same. Only the meaning of the bit patterns has changed. In the first example, we assumed that the bit patterns represented unsigned numbers and the adder produced the proper unsigned result. In the second example, we assumed that the bit patterns represented signed numbers. Again, the adder produced the proper signed result.



This proves a very important point. The adder in the ALU always adds bit patterns as if they are unsigned binary numbers. It is our interpretation of these bit patterns that decides if unsigned or signed numbers are indicated. The beauty of two's complement is that the bit patterns can be interpreted either way. This allows us to work with either signed or unsigned numbers without requiring different circuits for each.

Two's complement arithmetic also simplifies the arithmetic logic unit in another way. All microprocessors have a subtract instruction. Thus, the ALU must be able to subtract one number from another. However, if this required a separate subtraction circuit, the complexity and cost of the ALU would be increased. Fortunately, two's complement arithmetic allows the ALU to perform a subtract operation using an adder circuit. That is, the MPU uses the same circuit for both addition and subtraction.

The MPU performs subtraction by a binary addition process. To see why this works, it may be helpful to look at a similar process with the decimal number system. The decimal equivalent of two's complement is called ten's complement. Since you are more familiar with the decimal number system, briefly examine ten's complement arithmetic.

## Ten's Complement Arithmetic

An easy way to illustrate ten's complement is to consider an analogy. Visualize the odometer or mileage indicator on your car. Generally, this is a six-digit device that indicates mileage between 00,000.0 and 99,999.9 miles. Let's ignore the tenths digit and concentrate on the other five.


In an automobile, the register generally operates in only one direction (forward). However, consider what happens if it is turned backwards instead. Starting at +3 miles, the count proceeds backwards as follows:

00,003  
00,002  
00,001  
00,000  
99,999  
99,998  
99,997  
etc.

It is easy to visualize that 99,999 represents -1 mile. Also, 99,998 represents -2 miles; 99,997 represents -3 miles; etc. This is how signed numbers are represented in ten's complement form.

Once you accept this system for representing positive and negative numbers, you can perform arithmetic with these signed numbers. For example, if you add +3 and -2, the result should be +1. Using the system developed above, +3 is represented by 00003 while -2 is represented by 99,998. Thus, the addition looks like this:

$$\begin{array}{r}
 00003 \quad +3 \\
 +99998 \quad -2 \\
 \hline
 100001 \quad +1
 \end{array}$$

 Discard final carry.

If you now discard the final carry on the left in the sum, the answer is 00001, the representation of +1. You can also find the ten's complement of a digit by subtracting the digit from ten. For example, the ten's complement of 6 is 4 since  $10 - 6 = 4$ . To complement a number containing more than one digit, raise ten to a power equal to the total number of digits, then subtract the number from it. For example, to obtain the ten's complement of  $654_{10}$ , first raise ten to the third power since there are three digits in the number. Then, subtract 654 from the result.


$$\begin{array}{r}
 10^3 = 1000 \\
 -654 \\
 \hline
 346
 \end{array}$$

Thus, the ten's complement of  $654_{10}$  is  $346_{10}$ .

Once you find the ten's complement, you can subtract one number from another by an indirect method using only addition. Since childhood you have subtracted like this:

$$\begin{array}{r}
 \text{Minuend:} \quad 973 \\
 \text{Subtrahend:} \quad -654 \\
 \hline
 \text{Difference:} \quad 319
 \end{array}$$

However, you can arrive at the same answer by using the ten's complement of the subtrahend and adding. Recall that the ten's complement of  $654_{10}$  is  $346_{10}$ . Let's compare these two methods of subtraction:

<u>STANDARD METHOD</u>		<u>TEN'S COMPLEMENT METHOD</u>	
Minuend	973	973	Minuend
Subtrahend	<u>-654</u>	<u>+346</u>	Ten's complement of subtrahend
Difference	319	1319	Difference
		 Discard final carry	

Notice that when you use the ten's complement method, the answer is too large by  $1000_{10}$ . However, you can still arrive at the correct answer by simply discarding the final carry.

While the ten's complement method of subtraction works, it is not used because it is more complex than the standard method. In fact, it does not eliminate subtraction entirely since the ten's complement itself is found by subtraction.

The binary equivalent of ten's complement is two's complement. It overcomes the disadvantage of ten's complement in that the two's complement can be formed without any subtraction at all. Recall that you can form the two's complement of a binary number by changing all 0's to 1's, all 1's to 0's and then adding 1. Let's examine two's complement arithmetic in more detail.

## Two's Complement Subtraction


As in ten's complement arithmetic, you can form the two's complement by subtracting from a power of the base (two). However, because the MPU cannot subtract directly, it uses the method given earlier for finding the two's complement. Once the two's complement is formed, the MPU can perform subtraction indirectly by adding the two's complement of the subtrahend to the minuend.

To illustrate this point, look at the following two ways of subtracting  $26_{10}$  from  $69_{10}$ . The two numbers are expressed as they would appear to an 8-bit microprocessor. The standard method of subtraction looks like this:

Minuend:	01000101 <sub>2</sub>	69
Subtrahend:	<u>-00011010<sub>2</sub></u>	<u>-26</u>
Difference:	00101011 <sub>2</sub>	43

While this method works fine on paper, it's of little use to the microprocessor since the MPU has no subtract circuitry. However, the MPU can still perform subtraction by the indirect method of adding the two's complement of the subtrahend to the minuend:

	Minuend:	01000101
Two's complement of	Subtrahend:	+11100110
	Difference:	100101011

 Discard final carry

This illustrates a major reason for using the two's complement system to represent signed numbers. It allows the MPU to perform subtraction and addition with the same circuit.

The method that the MPU uses to perform subtraction is of little importance to the user of microprocessors. Most microprocessors have a subtract instruction. This instruction is used like any other without regard for how the operation is implemented internally. When the subtract instruction is implemented, the MPU automatically takes care of operations like complementing the subtrahend, adding, and discarding the carry. The procedure has been explained here so you can appreciate the importance of two's complement arithmetic.

## Arithmetic With Signed Numbers

There are many applications in which the microprocessor must work with signed numbers. In these cases, signed numbers are represented in two's complement form. While this greatly simplifies the circuitry of the MPU, it places an extra burden on the user. The programmer must ensure that all signed numbers are entered into the microprocessor in two's complement form. Also, the resulting data produced by the MPU may be in two's complement form. Here's how an 8-bit MPU handles signed numbers.

**Adding Positive Numbers.** Assume that the MPU is to add the two positive numbers +7 and +3. Since an 8-bit MPU is assumed, the arithmetic operation looks like this:

00000111	+ 7
+00000011	+ 3
00001010	+10

The sign bits are underlined. Remember, when representing signed numbers, that the MSB is the sign bit. A 0 represents “+” and a 1 represents “-.” In this example, you added +7 and +3 to form a sum of +10<sub>10</sub>. You know that all three numbers are positive since the MSB’s are all 0’s.

While this operation seems straightforward enough, it is easy for the unwary to make an error when adding positive numbers. Remember, the highest 8-bit positive number you can represent in two’s complement form is +127<sub>10</sub>. If the sum exceeds this value, an error occurs. For example, suppose you attempt to add +65<sub>10</sub> to +67<sub>10</sub>. The MPU adds the numbers as if they are unsigned binary:

$$\begin{array}{r} \underline{0}1000001 \\ \underline{0}1000011 \\ \hline 10000100 \end{array}$$

If the answer is interpreted as a two’s complement number, an error has occurred. You have added two positive numbers and yet the answer appears to be negative since the MSB of the sum is 1. This is called a two’s complement overflow. It occurs when the sum exceeds +127<sub>10</sub>. Many microprocessors have a way of detecting this condition. We will discuss this in more detail in a future unit.

**Adding Positive and Negative Numbers.** The real beauty of the two’s complement system is illustrated when you add numbers with unlike signs. For example, assume that an 8-bit microprocessor is to add +7 and -3. Remember, since these are signed numbers, they must be represented in two’s complement form. That is, +7 is represented as 00000111<sub>2</sub> while -3 is represented as 11111101<sub>2</sub>. If these two numbers are added, the sum will be:

$$\begin{array}{rcl} \text{Addend:} & 00000111 & (+7) \\ \text{Augend:} & +11111101 & +(-3) \\ \text{Sum:} & \underline{100000100} & (+4) \end{array}$$

↑  
Discard final carry

Notice that the sum is correct if you ignore the final carry bit. Keep in mind that the MPU adds the two numbers as if they were unsigned binary numbers. It is merely our interpretation of the answer that makes the system work for signed numbers.

The system also works when the negative number is larger. For example, when  $-9$  is added to  $+8$  the result should be  $-1$ . Remember, the signed numbers must be represented in two's complement form:

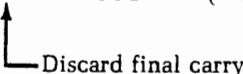
Addend:	11110111	(-9)
Augend:	00001000	+(+8)
Sum:	<u>11111111</u>	-1

Notice that the sum is the two's complement representation for  $-1$ .

**Adding Negative Numbers.** The final case involves two negative numbers. If both numbers are negative, then the sum should also be negative.

For example, suppose the MPU is to add  $-3$  to  $-4$ . Obviously, the result should be  $-7$ . The two signed numbers must be represented in two's complement form. That is,  $-3$  must be represented as  $11111101_2$  while  $-4$  must be represented as  $11111100_2$ . The MPU adds these two bit patterns as if they were unsigned binary numbers. Thus the result is:



Addend:	11111101	(-3)
Augend:	+11111100	+(-4)
Sum:	<u>111111001</u>	(-7)


 Discard final carry

Once again, the answer is correct if you ignore the final carry bit.

When you add two negative numbers, you must remember the capacity of the MPU. The largest negative number that can be represented by eight bits is  $-128_{10}$ . If the sum exceeds this value, the sum will appear to be in error. For example, suppose you add  $-120_{10}$  to  $-18_{10}$ .

	10001000	(-120)
	11101110	+(-18)
	<u>101110110</u>	


 Ignore carry
 
 Sign bit

Notice that the sign bit in the sum is 0, representing a positive number. Thus, the MPU has added two negative numbers and has produced a positive result. This apparent error is caused by exceeding the 8-bit capacity. This is another example of two's complement overflow.

## Self-Test Review

12. In microprocessors, signed numbers are represented in \_\_\_\_\_ form.
13. The ALU adds bit patterns as if they represent \_\_\_\_\_ binary numbers.
14. When a microprocessor executes a subtract instruction, what operations are actually performed inside the MPU?
15. What is the largest 8-bit positive number that can be represented in two's complement form?
16. When you are adding two positive numbers, what is meant by two's complement overflow?
17. If  $+19_{10}$  and  $-21_{10}$  are added by an 8-bit microprocessor, the two's complement result will be \_\_\_\_\_.
18. Can two's complement overflow occur when two negative numbers are added?
19. A microprocessor adds  $10001110_2$  to  $00010001_2$ . If these are unsigned binary numbers, the resulting bit pattern will be \_\_\_\_\_. If these are two's complement numbers, the resulting bit pattern will be \_\_\_\_\_.
20. If the bit patterns in question 19 represent unsigned numbers, the resulting bit pattern represents decimal \_\_\_\_\_.
21. If the bit patterns in question 19 represent two's complement numbers, the resulting bit pattern represents decimal \_\_\_\_\_.

## Answers

12. Two's complement.
13. Unsigned.
14. The following operations occur:
  1. The MPU complements the subtrahend by changing 0's to 1's and 1's to 0's.
  2. One is added to the complemented subtrahend to form the two's complement.
  3. The two's complement of the subtrahend is added to the minuend.
15.  $01111111_2$  or  $+127_{10}$ .
16. When you add positive numbers, two's complement overflow occurs when the sum exceeds  $+127_{10}$ .
17.  $11111110_2$  or  $-2_{10}$ .
18. Yes. When you add negative numbers, two's complement overflow occurs when the sum exceeds  $-128_{10}$ .
19. In either case, the resulting bit pattern will be 10011111.
20.  $159_{10}$ .
21.  $-97_{10}$ .



## BOOLEAN OPERATIONS

Along with the basic mathematical processes examined earlier, the microprocessor can manipulate binary numbers logically. This system was conceived using the theorems developed by the mathematician George Boole. As a result, this branch of binary mathematics is given the name Boolean Algebra. In this section, the Boolean operations performed by the microprocessor will be examined. A more detailed description of Boolean Algebra is provided in the Heathkit Continuing Education Series course titled "Digital Techniques."

### AND Operation

The AND function produces the logical product of two or more logic variables. That is, the logical product of an AND operation is logic 1 if all of the variable inputs are logic 1. If any of the input variables are logic 0, the logical product is 0. This process can be represented by the formula  $A \cdot B = C$ , where A and B represent input variables (logic 1 or 0) and C represents the output or logical product of the AND operation. The AND function is designated by a dot between the variables. Do not confuse it with the mathematical multiplication sign.

Figure 3-6 is a "truth" table for a two-variable AND function. The 1's and 0's represent all of the possible logic combinations. Thus, you can see that the AND function is a sort of "all or nothing" operation. Unless all the input variables are logic 1, the output cannot be logic 1.

INPUTS		OUTPUT
A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

Figure 3-6  
Truth Table for an AND function.

When the microprocessor implements the logic AND operation, one 8-bit binary number is ANDed with a second 8-bit binary number. Refer to Figure 3-7 for an illustration of this process.

<u>8-BIT</u> <u>NUMBER</u>			<u>8-BIT</u> <u>NUMBER</u>		<u>RESULT OF</u> <u>AND OPERATION</u>
MSB	1	•	1	=	1 MSB
	0	•	0	=	0
	0	•	1	=	0
	1	•	0	=	0
	1	•	1	=	1
	0	•	1	=	0
	1	•	0	=	0
LSB	0	•	0	=	0 LSB

Figure 3-7  
8-bit logic AND operation.

Although more than two logic variables can be ANDed together, the microprocessor operates on only two variables at a time. Now try one more example of the AND operation. AND 10011101 with 11000110.

$$\begin{array}{rcl}
 1 \cdot 1 & = & 1 \quad \text{MSB} \\
 0 \cdot 1 & = & 0 \\
 0 \cdot 0 & = & 0 \\
 1 \cdot 0 & = & 0 \\
 1 \cdot 0 & = & 0 \\
 1 \cdot 1 & = & 1 \\
 0 \cdot 1 & = & 0 \\
 1 \cdot 0 & = & 0 \quad \text{LSB}
 \end{array}$$

## OR Operation

The OR (sometimes known as inclusive OR) function produces the logical sum of two or more logic variables. That is, the logical sum of an OR operation is logic 1 if either input is logic 1. The logical sum is 0 if **all** of the input variables are logic 0. This process can be represented by the formula  $A + B = C$ , where A and B represent input variables and C represents the output or logical sum of the OR operation. The OR function is designated by a plus sign (or a circled dot  $\odot$ ) between the variables. Do not confuse the plus sign with the mathematical add sign.

Figure 3-8 is a “truth” table for a two-variable OR function. The 1’s and 0’s represent all of the possible logic combinations. Thus, you can see that the OR function is a sort of “either or both” operation. If either or both input variables are logic 1, the output must be logic 1.

INPUTS		OUTPUT
A	B	C
0	0	0
1	0	1
0	1	1
1	1	1

Figure 3-8  
Truth Table for an OR function.

When the microprocessor implements the logic OR operation, one 8-bit binary number is ORed with a second 8-bit binary number. Refer to Figure 3-9 for an illustration of this process.

<u>8-BIT</u> <u>NUMBER</u>			<u>8-BIT</u> <u>NUMBER</u>		<u>RESULT OF</u> <u>OR OPERATION</u>	
MSB	1	+	1	=	1	MSB
	0	+	0	=	0	
	0	+	1	=	1	
	1	+	0	=	1	
	1	+	1	=	1	
	0	+	1	=	1	
	1	+	0	=	1	
LSB	0	+	0	=	0	LSB

Figure 3-9  
8-bit logic OR operation.

As with the AND function, two or more logic variables can be ORed together. However, the microprocessor operates on only two variables at a time. Now try one more example of the OR operation. OR 10011101 with 11000101.

$$\begin{array}{rcl}
 1 + 1 & = & 1 \quad \text{MSB} \\
 0 + 1 & = & 1 \\
 0 + 0 & = & 0 \\
 1 + 0 & = & 1 \\
 1 + 0 & = & 1 \\
 1 + 1 & = & 1 \\
 0 + 0 & = & 0 \\
 1 + 1 & = & 1 \quad \text{LSB}
 \end{array}$$

## Exclusive OR Operation

The Exclusive OR (EOR or XOR) function performs a logical test for "equalness" between two logic variables. That is, if two variable inputs are equal (both logic 1 or 0), the output or result of the EOR operation is logic 0. If the inputs are not equal (one is logic 1, the other logic 0) the output is logic 1. This can be represented by the formula  $A \oplus B = C$ , where A and B represent input variables and C represents the output or result. The EOR function is designated by a circled plus sign between the variables.

Figure 3-10 is a "truth" table for the EOR function. The 1's and 0's represent all of the possible logic combinations. You can see that the EOR function is a sort of "either but not both" operation. That is, either input can be logic 1 or 0, but not both for a logic 1 output.

INPUTS		OUTPUT
A	B	C
0	0	0
0	1	1
1	0	1
1	1	0

Figure 3-10

Truth Table for an EOR function.

When the microprocessor implements the logic EOR operation, one 8-bit binary number is exclusively ORed with a second 8-bit number. Refer to Figure 3-11 for an illustration of this process.

<u>8-BIT</u> <u>NUMBER</u>			<u>8-BIT</u> <u>NUMBER</u>		<u>RESULT OF</u> <u>EOR OPERATION</u>	
MSB	1	$\oplus$	1	=	0	MSB
	0	$\oplus$	0	=	0	
	0	$\oplus$	1	=	1	
	1	$\oplus$	0	=	1	
	1	$\oplus$	1	=	0	
	0	$\oplus$	1	=	1	
	1	$\oplus$	0	=	1	
LSB	0	$\oplus$	0	=	0	LSB

Figure 3-11  
8-bit logic EOR operation.

Now try one more example of the EOR operation. EOR  $10011101_2$  with  $11000101_2$ .

$$\begin{array}{rcl}
 1 \oplus 1 & = & 0 \quad \text{MSB} \\
 0 \oplus 1 & = & 1 \\
 0 \oplus 0 & = & 0 \\
 1 \oplus 0 & = & 1 \\
 1 \oplus 0 & = & 1 \\
 1 \oplus 1 & = & 0 \\
 0 \oplus 0 & = & 0 \\
 1 \oplus 1 & = & 0 \quad \text{LSB}
 \end{array}$$

## Invert Operation

The invert operation performs a direct complement of a single input variable. That is, a logic 1 input will produce a logic 0 output. This process can be represented by the truth table in Figure 3-12.

INPUT	OUTPUT
A	$\overline{A}$
1	0
0	1

Figure 3-12

Truth Table for an invert function.

Note that the complement of A is  $\overline{A}$ . The bar above the A indicates that A has been inverted, and is read "not A." Therefore, the complement of A is "not A" ( $\overline{A}$ ).

When the microprocessor implements the logic invert operation, the 8-bit binary number is complemented. This operation is also known as 1's complement. Thus, the complement of  $11010110_2$  is  $00101001_2$ . As with the previous logic operations, the invert function operates on each individual bit of the 8-bit number.

## Self-Test Review

22. The result of an AND operation is binary 1 when:
- A. All inputs are binary 0.
  - B. Any one input is binary 0.
  - C. All inputs are binary 1.
  - D. Any one input is binary 1.
23. Perform the AND operation on the following 8-bit number pairs.
- A. 11010110 and 10000111.
  - B. 00110011 and 11110000.
  - C. 10101010 and 11011011.
24. The result of an OR operation is binary 0 when:
- A. All inputs are binary 1.
  - B. All inputs are binary 0.
  - C. Any one input is binary 1.
  - D. Any one input is binary 0.
25. Perform the OR operation on the following 8-bit number pairs.
- A. 11010110 and 10000111.
  - B. 00110011 and 11110000.
  - C. 10101010 and 11011011.
26. The result of an XOR operation is binary 0 if the inputs are:
- A. Equal.
  - B. Not equal.

27. The symbol for the EOR operation is:
- A.  $\cdot$
  - B.  $+$
  - C.  $\oplus$
  - D.  $\times$
28. Perform the EOR operation on the following 8-bit number pairs.
- A. 11010110 and 10000111.
  - B. 00110011 and 11110000.
  - C. 10101010 and 11011011.
29.  $\overline{A}$  represents the \_\_\_\_\_ of A.
- A. Sum.
  - B. Product.
  - C. Complement.
  - D. Supplement.
30. Perform the invert operation on the following 8-bit numbers.
- A. 11010110.
  - B. 00110011.
  - C. 10101010.



## Answers

22. C. All inputs are binary 1.

23. A.  $1 \cdot 1 = 1$   
 $1 \cdot 0 = 0$   
 $0 \cdot 0 = 0$   
 $1 \cdot 0 = 0$   
 $0 \cdot 0 = 0$   
 $1 \cdot 1 = 1$   
 $1 \cdot 1 = 1$   
 $0 \cdot 1 = 0$

B. 00110000.

C. 10001010.

24. B. All inputs are binary 0.

25. A.  $1 + 1 = 1$   
 $1 + 0 = 1$   
 $0 + 0 = 0$   
 $1 + 0 = 1$   
 $0 + 0 = 0$   
 $1 + 1 = 1$   
 $1 + 1 = 1$   
 $0 + 1 = 1$

B. 11110011.

C. 11111011.

26. A. Equal.

27. C.  $\oplus$
28. A.  $1 \oplus 1 = 0$   
 $1 \oplus 0 = 1$   
 $0 \oplus 0 = 0$   
 $1 \oplus 0 = 1$   
 $0 \oplus 0 = 0$   
 $1 \oplus 1 = 0$   
 $1 \oplus 1 = 0$   
 $0 \oplus 1 = 1$
- B. 11000011.
- C. 01110001.
29. C. Complement.
30. A. 00101001.
- B. 11001100.
- C. 01010101.

## EXPERIMENT 4

Perform Experiment 4 in Unit 9 of this course. After you finish the experiment, return to this Unit and complete the Unit Examination.

## UNIT EXAMINATION

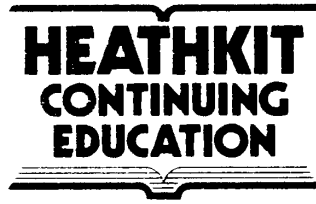
1. Add  $10010110_2$  to  $1101_2$ .
2. Subtract  $1011_2$  from  $10110110_2$ .
3. Multiply  $1001_2$  by  $1100_2$ .
4. Divide  $100111_2$  with  $110_2$ .
5. The 1's complement of  $00110110_2$  is \_\_\_\_\_.
6. The 2's complement of  $00110110_2$  is \_\_\_\_\_.
7. Using 2's complement arithmetic, add  $+75_{10}$  to  $-6_{10}$ .
8. Using 2's complement arithmetic, add  $-35_{10}$  to  $-75_{10}$ .
9. Using 2's complement arithmetic, subtract  $-15_{10}$  from  $-85_{10}$ .
10. The truth table Figure 3-13 represents the logical \_\_\_\_\_ function.

INPUT		OUTPUT
A	B	C
0	0	0
1	0	1
0	1	1
1	1	0

Figure 3-13  
Truth Table for Exam Question 10.

11. Logically AND  $11011010$  with  $10010110$ .
12. Logically OR  $11011010$  with  $10010110$ .
13. Logically EOR  $11011010$  with  $10010110$ .
14. Logically invert  $11011010$ .





# Individual Learning Program

## MICROPROCESSORS

### *Unit 4*

## INTRODUCTION TO PROGRAMMING

EE-3401

HEATH COMPANY  
BENTON HARBOR, MICHIGAN 49022

Copyright © 1977  
Heath Company  
All Rights Reserved  
Printed in the United States of America

## CONTENTS

Introduction .....	4-3
Unit Objectives .....	4-4
Unit Activity Guide .....	4-5
Branching .....	4-6
Conditional Branching .....	4-20
Algorithms .....	4-29
Additional Instructions .....	4-46
Experiments .....	4-58
Unit Examination .....	4-59
Examination Answers .....	4-61

## *Unit 4*

# INTRODUCTION TO PROGRAMMING

## INTRODUCTION

In the final analysis there are only two things you can do with a microprocessor. You can program it and you can interface it with the outside world. In this course, you learn to program the microprocessor first. This unit, along with the associated cassette tape and experiments, will serve as an introduction to programming.

The programs you encounter in this unit are simple enough that anyone can understand them, and yet they illustrate many important concepts. By studying these programs, you will develop an understanding of how the microprocessor handles complex tasks. At the same time, you will gain practice using the instruction set.

## UNIT OBJECTIVES

When you have completed this unit, you will be able to:

1. Explain the difference between machine language, assembly language, interpretive language, and compiler language.
2. Define assembler, compiler, interpreter, object program, source program, BASIC, FORTRAN, and COBOL.
3. Draw the symbols used in flow charting and explain the purpose of each.
4. Develop flow charts that illustrate step-by-step procedures for solving simple problems.
5. Explain the purpose of conditional and unconditional branching.
6. Using the block diagram of the hypothetical microprocessor, trace the data flow during the execution of a branch instruction.
7. Compute the proper relative address for branching forward or backward from one point to another in a program.
8. Explain the purpose of the carry, negative, zero, and overflow flags. Give an example of a situation that can cause each to be set and another example that will cause each to clear. List eight instructions that test one of these flags.
9. Write programs that can: multiply by repeated addition; divide by repeated subtraction; convert binary to BCD; convert BCD to binary; add multiple-precision numbers; subtract multiple-precision numbers; add BCD numbers.



## UNIT ACTIVITY GUIDE

	Completion Time
<input type="checkbox"/> Play Cassette Tape Section "Introduction to Programming."	_____
<input type="checkbox"/> Read Section on Branching.	_____
<input type="checkbox"/> Complete Self-Test Review Questions 1-9.	_____
<input type="checkbox"/> Read Section on Conditional Branching.	_____
<input type="checkbox"/> Complete Self-Test Review Questions 10-19.	_____
<input type="checkbox"/> Read Section on Algorithms.	_____
<input type="checkbox"/> Complete Self-Test Review Questions 20-29.	_____
<input type="checkbox"/> Read Section on Additional Instructions.	_____
<input type="checkbox"/> Complete Self-Test Review Questions 30-37.	_____
<input type="checkbox"/> Perform Programming Experiments 5 and 6.	_____
<input type="checkbox"/> Complete Unit Examination.	_____
<input type="checkbox"/> Check Examination Answers.	_____

## BRANCHING

The programs discussed earlier were all “straight line” programs; the instructions were executed one after another in the order in which they were written. Programs of this type are extremely limited because they use only a fraction of the microprocessor’s power.

The real power of the microprocessor comes from its ability to execute a section of a program over and over again. In an earlier program we saw that two numbers could be multiplied by repeated addition. As long as the numbers are very small and we know the value of the two numbers, we can write a “straight line” program to multiply the numbers. For example, 9 could be multiplied by 4 with the following program:

Address	Instruction/Data	Comments
00	LDA 05	Load direct
01	ADD 05	Add direct
02	ADD 05	Add direct
03	ADD 05	Add direct
04	HLT	
05	09	

This technique is very crude for a number of reasons. If the two numbers are large, such as 98 and 112, the number of ADD instructions becomes excessive. Moreover, the values of the two numbers to be multiplied are generally not known. Therefore, even if we were willing to write enough ADD instructions, we simply would not know how many to write. Obviously, some better technique must be available.

A technique that is used in virtually every program is called **looping**. This allows a section of the program to be run as often as needed. Every microprocessor has a group of instructions called JUMP or BRANCH instructions that allow it to execute these program loops. These allow the microprocessor to escape the normal instruction sequence.

The microprocessor discussed in this course has both jump and branch instructions. In this unit, we will confine our discussion to the branch instructions. In a later unit we will discuss the jump instructions.

Before discussing the types of branch instructions, we must first discuss a new addressing mode called relative addressing.

## Relative Addressing

In previous units, we discussed immediate addressing and direct addressing. Recall that in the immediate addressing mode no address is specified. The data is assumed to be the byte following the opcode. In direct addressing, an address is given. The data is assumed to be at that address.

Branch instructions are somewhat different from the instructions discussed earlier. While the branch instruction has an address associated with it, the address does not indicate the location of data. Instead, the address indicates the location of the next instruction that is to be executed.

The format of the branch instruction is shown in Figure 4-1. All branch instructions are 2-byte instructions. The first byte is the 8-bit opcode. This code identifies the particular type of branch instruction. As you will see later, a microprocessor may have a dozen or more different branch instructions. Each has its own opcode that uniquely identifies it.

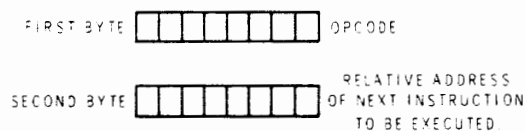


Figure 4-1  
Format of the branch instruction.

The second byte of the branch instruction indicates the point to which the program is to branch. That is, it specifies the address of the next instruction that is to be executed.

In some microprocessors, the address is absolute. That is, the address is the memory location that holds the next instruction. In this case, the instruction `BRANCH 3016` would mean that the instruction to be executed next is at address 30<sub>16</sub>. In other words, some microprocessors use direct addressing when branching.

Our hypothetical microprocessor uses a different technique called relative addressing. In this addressing mode, the byte following the opcode does not represent an absolute address. Instead, it is a number that must be added to the program counter to form the new address. Consider the instruction:

`BRANCH 3016`

Using relative addressing, this does not mean that the next instruction is to be taken from memory location  $30_{16}$ . Rather, it means that  $30_{16}$  must be added to the present contents of the program counter. Thus, if the program counter is at  $08_{16}$  when the BRANCH  $30_{16}$  instruction is executed, the next instruction will be fetched from location  $08_{16} + 30_{16} = 38_{16}$ .

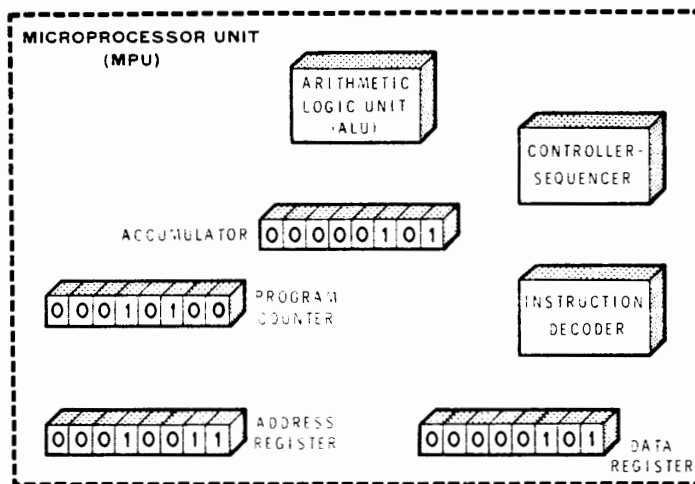
By the same token, if the contents of the program counter is  $FA_{16}$  when a BRANCH  $03$  is encountered, the next instruction will be fetched from location  $FA_{16} + 03 = FD_{16}$ . Notice that this allows the MPU to jump over the instructions at addresses  $FB_{16}$  and  $FC_{16}$ .

## Executing a Branch Instruction

Determining the relative address to use as the second byte of the branch instruction can be confusing unless you keep in mind the method by which the MPU executes a program. Therefore, let's go through the manipulations that take place within the MPU during the execution of the branch instruction.

Figure 4-2 shows sections of a program stored in memory. Let's assume that the MPU has been executing this program. Let's further assume that the MPU just completed the execution of the LDA  $05$  instruction at addresses  $12_{16}$  and  $13_{16}$ . The address register still holds the address of the last byte that was read from memory. The accumulator and data register hold the contents ( $05$ ) of the last location that was read out.

Notice that the program counter contains the address of the next instruction to be executed. This address points to the branch instruction in memory location  $14_{16}$ . Let's pick up the action at this point.



MEMORY		
ADDRESS	BINARY CONTENTS	MNEMONICS/ CONTENTS
0001 0010	0000 0110	LDA
0001 0011	0000 0101	05 <sub>16</sub>
0001 0100	0010 0000	SRA
0001 0101	0000 0111	07 <sub>16</sub>
0001 0110	_____	_____
0001 1100	_____	_____
0001 1101	1000 1011	ADD
0001 1110	0000 0110	06 <sub>16</sub>
0001 1111	_____	_____

**Figure 4-2**  
Status of the MPU registers after  
executing the LDA 05 instruction.

Figure 4-3 shows how the first byte of the branch instruction is fetched. This is the standard fetch operation that was discussed earlier:

1. The address ( $14_{16}$ ) is transferred from the program counter to the address register.
2. The program counter is incremented to  $15_{16}$ .
3. The address is strobed onto the address bus.
4. The contents of the selected memory location are transferred via the data bus to the data register.
5. The instruction decoder examines this opcode and finds it to be a branch instruction.

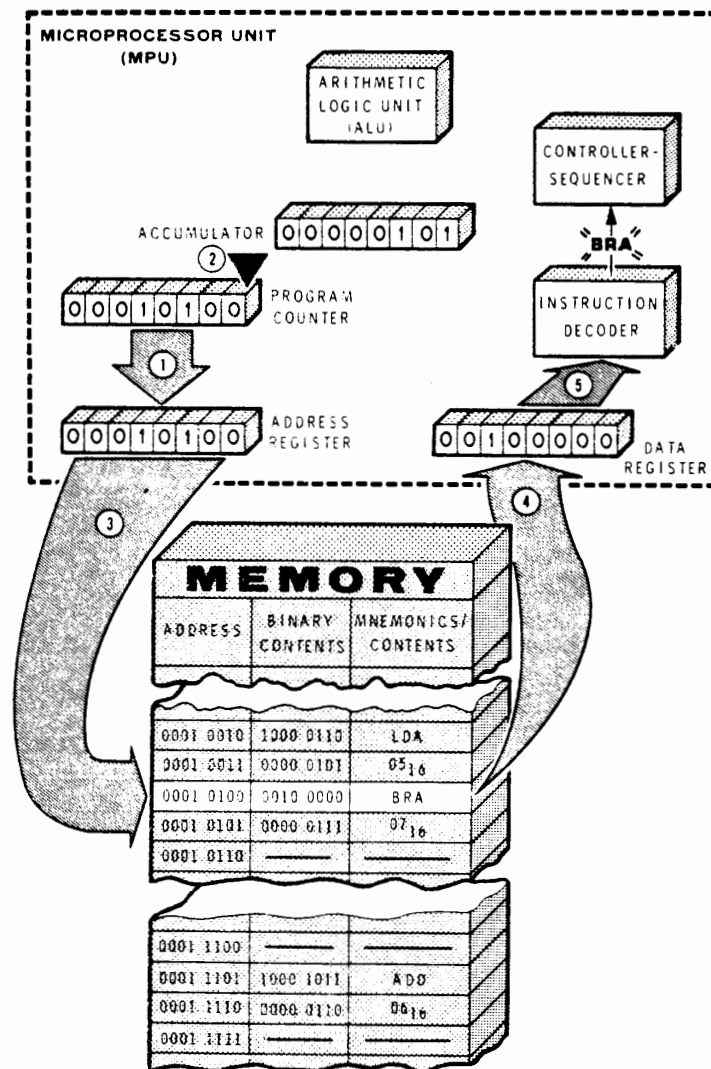


Figure 4-3  
Fetching the BRA instruction.

Therefore, the controller-sequencer starts the procedure for executing a branch instruction.

During the next machine cycle, the relative address is fetched. This procedure is shown in Figure 4-4. The major events are:

1. The address ( $15_{16}$ ) is transferred from the program counter to the address register.
2. The program counter is incremented to  $16_{16}$ .
3. The address ( $15_{16}$ ) is strobed onto the address bus.
4. The contents of location  $15_{16}$  are transferred to the data register via the data bus.

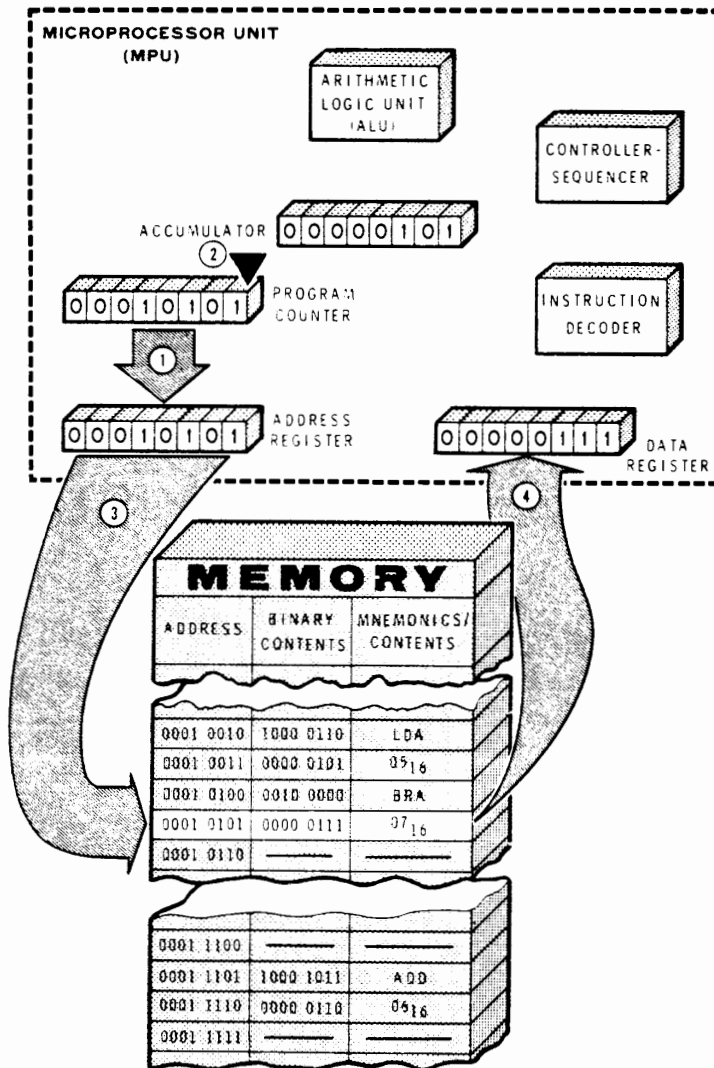


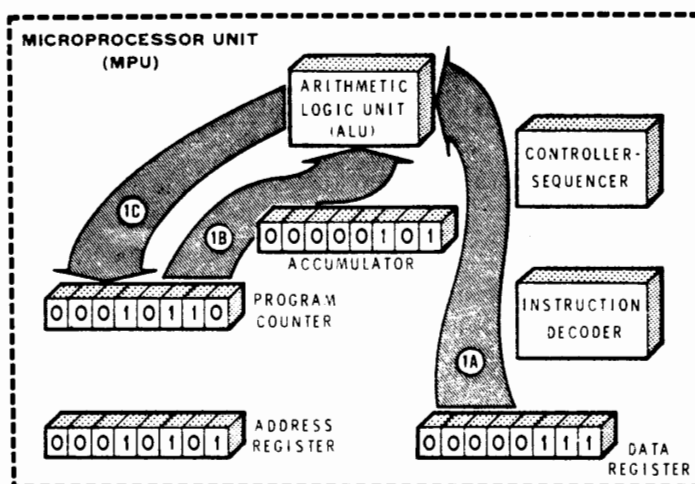
Figure 4-4  
Fetching the relative address.

Figure 4-5 shows the state of the various registers after the relative address is fetched. The relative address ( $07_{16}$ ) is in the data register. Now look at the program counter. Notice that it points to address  $16_{16}$ . However, the MPU has not yet finished executing the branch instruction. It must now compute the new address by adding the relative address to the program count. It uses the addition capabilities of the ALU to perform this function. That is, the program count and relative address are strobed into the ALU. The ALU adds the two together and produces a sum of

0001	0110	program count
<u>0000</u>	<u>0111</u>	relative address
0001	1101	new program count

This sum is loaded into the program counter. Thus, the next instruction is fetched from memory location  $1D_{16}$ . That is, the next instruction to be executed is the  $ADD\ 06_{16}$  instruction.





MEMORY		
ADDRESS	BINARY CONTENTS	MNEMONICS/ CONTENTS
0001 0010	1000 0110	LDA
0001 0011	0000 0101	05 <sub>16</sub>
0001 0100	0010 0000	BRA
0001 0101	0000 0111	07 <sub>16</sub>
0001 0110	—	—
0001 1100	—	—
0001 1101	1000 1011	ADD
0001 1110	0000 0110	06 <sub>16</sub>
0001 1111	—	—

**Figure 4-5**  
Computing the address of the  
next instruction.

## Branching Forward

Branching in the forward direction is a simple task if you know the value of the program count when the relative address is added. A couple of examples will illustrate the procedure.

In Figure 4-6A, the BRANCH 03 instruction is placed in locations 32<sub>16</sub> and 33<sub>16</sub>. Assuming this instruction is executed, from which location will the next instruction be fetched? Remember that the program counter will always point to the next byte in sequence. Since the last byte fetched was the relative address from location 33<sub>16</sub>, the program counter must be at 34<sub>16</sub> when the relative address is added. Adding the relative address produces a new program count of

$$\begin{array}{r} 34_{16} \\ + 3_{16} \\ \hline 37_{16} \end{array}$$

Thus, the next instruction will be fetched from location 37<sub>16</sub>.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ HEX CONTENTS
32	20	BRA
33	03	03
34	—	—
35	—	—
36	—	—
37	—	—
38	—	—

A

Program will  
branch to here

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ HEX CONTENTS
18	20	BRA
19	??	??
1A	Originating Address	
1B		
1C		
1D		
1E		
1F		
20		
21	Destination Address	
22		
23		
24		

B

We wish to  
Branch to here

Figure 4-6  
Branching forward.

Figure 4-6B shows a slightly different situation. Here we wish to branch to the instruction at address  $24_{16}$ . The opcode for the branch instruction is at address  $18_{16}$ . What relative address is required at location  $19_{16}$  in order to implement this branch?

Keep in mind that the program count will automatically advance to  $1A_{16}$  after the relative address is fetched from address  $19_{16}$ . Also, remember that the relative address is added to the program count. Thus, a relative address of 00 would result in a "branch" to location  $1A_{16}$ . A relative address of 01 would result in a branch to location  $1B_{16}$ . Continuing this procedure until location  $24_{16}$  is reached, you find that a relative address of  $10_{10}$  is required. That is, the relative address must be  $0A_{16}$  or  $10_{10}$ .

There is a simple procedure for determining the relative address when branching forward. Subtract the originating address from the destination address. The difference is the relative address.

In our example, the originating address is  $1A_{16}$ . Remember this is the program count at the time the relative address is added. The destination address or the address to which you wish to branch is  $24_{16}$ . Subtracting the originating address from the destination address, you find that the required relative address is

0010 0100 <sub>2</sub>	$24_{16}$	Destination address
<u>-0001 1010<sub>2</sub></u>	<u><math>1A_{16}</math></u>	Originating address
0000 1010 <sub>2</sub>	$0A_{16}$	Relative address

As you can see, a relative address of  $0A_{16}$  is called for.

## Branching Backward

A backward branch is used when a part of the program is to be repeated. The technique used for branching backward is similar to that used in branching forward. The difference is that a negative number is used as the relative address. As you learned earlier, two's complement representation is used to signify negative and positive numbers. Therefore, the relative address portion of any branch instruction is interpreted as a two's complement number.

This means that bit 7 of the relative address byte is a sign bit. A 0 at bit 7 tells the MPU to branch forward; a 1 tells it to branch backward. Thus, the positive values for the relative address extend from  $0000\ 0000_2$  to  $0111\ 1111_2$ . This is  $00_{16}$  to  $7F_{16}$  and  $00_{10}$  to  $+127_{10}$ .

The negative values extend from  $1111\ 1111_2$  to  $1000\ 0000_2$ . This is  $FF_{16}$  to  $80_{16}$  and  $-1$  to  $-128_{10}$ . But remember, the relative address is with respect to the present program count. At the time the relative address is added, the program count points to the next byte after the relative address. Let's look at two examples of branching backward.

The first example is shown in Figure 4-7A. To what point does the MPU branch when the branch instruction at address  $5D_{16}$  is executed? Notice that the relative address is  $F9_{16}$ . In binary this is  $1111\ 1001_2$ . Recall that this is the two's complement representation of  $-7$ . Thus, the program count should jump backwards 7 bytes — but from what point? Recall that after the byte at address  $5E$  is fetched, the program count will automatically advance to  $5F_{16}$  or  $0101\ 1111_2$ . When the relative address ( $F9_{16}$  or  $1111\ 1001_2$ ) is added, the result is

0101 1111	← Old program count
+ 1111 1001	← Relative Address
1 0101 1000	← New program count

↑

Carry is ignored —

The carry bit is ignored, leaving a new program count of  $58_{16}$ . Thus, the next instruction will be fetched from address  $58_{16}$ .

Figure 4-7B shows a different problem. Here we want the branch instruction at addresses B0 and B1 to direct the MPU back to address A0. What relative address is required? A simple procedure is:

1. Subtract the destination address from the originating address.
2. Take the two's complement of the difference.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ HEX CONTENTS
56	—	—
57	—	—
58	—	—
59	—	—
5A	—	—
5B	—	—
5C	—	—
5D	20	BRA
5E	F9	F9
5F	—	—

Program branches to here

**A**

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ HEX CONTENTS
A0	—	—
A1	—	—
A2	—	—
A3	—	—
A4	—	—
A5	—	—
A6	—	—
A7	—	—
A8	—	—
A9	—	—
AA	—	—
AB	—	—
AC	—	—
AD	—	—
AE	—	—
AF	—	—
B0	20	BRA
B1	??	??
B2	—	—

We wish to branch to here

**B**

Figure 4-7  
Branching backwards.

In our example, the program count will be advanced to  $B2_{16}$  after the relative address is fetched. This is our originating address. The point to which we wish to branch is  $A0_{16}$ . This is our destination address. Subtracting yields a difference of

1011 0010 <sub>2</sub>	$B2_{16}$	Originating address
– 1010 0000 <sub>2</sub>	$A0_{16}$	Destination address
0001 0010 <sub>2</sub>	$12_{16}$	Difference

Next you compute the relative address by taking the two's complement of the difference. The two's complement of  $0001\ 0010_2$  is  $1110\ 1110_2$ . In hexadecimal this is  $EE_{16}$ . Thus, the required relative address is  $EE_{16}$ .

## Self-Test Review

1. What addressing mode is used by the branch instruction?
2. What does the second byte of a branch instruction indicate?
3. What happens in the MPU during the execution of the branch instruction?
4. What type of relative address causes a branch forward?
5. What type of relative address causes a branch backwards?
6. What is the maximum number of memory locations that can be branched over during a forward branch?
7. What is the maximum number of memory locations that can be branched over during a backward branch?
8. The opcode for the branch instruction is at address  $20_{16}$ . The relative address is  $06_{16}$ . After the branch instruction is executed, from what address will the next opcode be fetched?
9. The opcode for the branch instruction is at address  $20_{16}$ . The relative address is  $F1_{16}$ . After the branch instruction is executed, from what address will the next opcode be fetched?

## Answers

1. Relative addressing.
2. The second byte of the branch instruction is the relative address. This number is added to the contents of the program counter to form the absolute address.
3. The relative address is retrieved from memory and is added to the program count. The new program count goes into the program counter.
4. A positive two's complement number.
5. A negative two's complement number.
6.  $0111\ 1111_2$  or  $+127_{10}$ .
7.  $1000\ 0000_2$  or  $-128_{10}$ .
8.  $28_{16}$ . Recall that during the execution of the branch instruction, the program counter will be incremented twice to  $22_{16}$ . Thus, when the relative address ( $06_{16}$ ) is added, the new address becomes  $28_{16}$ .
9. As in answer 8, the program counter is automatically advanced to  $22_{16}$  ( $0010\ 0010_2$ ) before the relative address is added.  $F1_{16}$  is equal to  $1111\ 0001_2$ . When this is added to the program count, the new address becomes

	0010 0010 <sub>2</sub>	Old program count
	1111 0001 <sub>2</sub>	Relative address
	<hr style="width: 100%; border: 0.5px solid black;"/>	
	1 0001 0011 <sub>2</sub>	New program count

↑  
Ignore carry

Thus, the next opcode will be fetched from address  $13_{16}$ .

## CONDITIONAL BRANCHING

The branch instruction allows the MPU to jump forward over a block of data or over a portion of a program. It also allows the MPU to jump backwards so a group of instructions can be repeated.

Until now we have been discussing the **unconditional** branch instruction. This type of instruction always results in a program branch. For this reason, it is called the **BR**anch **A**lways instruction. Its mnemonic is BRA.

There are other types of branch instructions that greatly expand the versatility of the MPU. These are called **conditional** branch instructions. Unlike BRA, these instructions cause a branch only if some specified condition is met.

A good example of a conditional branch instruction is the Branch If Minus (BMI). This instruction may or may not initiate a branch operation, depending on the result of some previous arithmetic or logic operation. This instruction might be placed after a subtract instruction. If the result of the subtraction is a negative number, the branch would be implemented. Otherwise, the MPU would continue to fetch and execute instructions in numerical order. An example may help to illustrate this.

Figure 4-8 shows part of a program that uses the branch if minus (BMI) instruction. Let's start with the instruction at address 95<sub>16</sub>. This instruction causes the contents of location B0<sub>16</sub> to be loaded into the accumulator. Next, the SUB instruction subtracts the contents of location B1<sub>16</sub> from the number in the accumulator. The next instruction (BMI) examines the result of the subtraction. If the result was a minus number, the program will branch over the next three bytes. That is, the next instruction to be executed is the STA instruction at address 9E<sub>16</sub>. Thus, the resulting number in the accumulator is stored in location B3<sub>16</sub> and the MPU halts.

If the result of the subtraction is not minus, the BMI instruction has no effect. That is, the BMI instruction is fetched and executed but no branch occurs because the specified condition is not met. In this case, the next instruction to be executed is the STA instruction at address 9B<sub>16</sub>. Thus, the result of the subtraction will be stored in location B2<sub>16</sub>.



HEX ADDRESS	HEX CONTENTS	MNEMONIC/HEX CONTENTS	COMMENTS
95	96	LDA	Load accumulator direct
96	B0	B0	with contents of this address.
97	90	SUB	Subtract
98	B1	B1	the contents of this address.
99	2B	BMI	If result is minus
9A	03	03	branch this far.
9B	97	STA	If result is not minus, store
9C	B2	B2	at this address;
9D	3E	HLT	then halt.
9E	97	STA	If result is minus, store
9F	B3	B3	it at this address;
A0	3E	HLT	then halt.

Figure 4-8

This program uses the BMI instruction to make a simple decision.

Notice that the program flow can take one of two paths, depending on the result of the subtraction. The BMI instruction gives the MPU this capability. The conditional branch instructions are sometimes called "decision making instructions." The reason for this becomes obvious if you consider the implications of our sample program. Here the MPU decides if the number at address  $B1_{16}$  is larger than that at  $B0_{16}$ . The program path is determined by the outcome of this decision. If the number in  $B1_{16}$  is larger, the result of the subtraction is a negative number. In this case, the result is stored in location  $B3_{16}$ . Otherwise, the resulting difference is stored in location  $B2_{16}$ .

Virtually all programs must make some type of decision. Some frequently encountered decisions are:

"Which of two numbers is larger?"

"Does this byte represent a letter of the alphabet or a numeral?"

"Are these two numbers equal?"

"Is this an even number?"

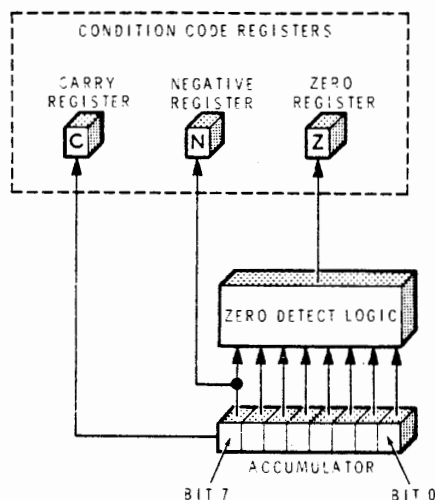
"Has the program loop been repeated the proper number of times?"

Conditional branch instructions are used in making all of these decisions.

## Condition Codes

As the name implies, a conditional branch instruction causes a program branch only if some specified condition is met. Some commonly monitored conditions are:

1. Did a previous operation result in a negative number in the accumulator?
2. Did a previous operation result in zero in the accumulator?
3. Did a previous operation result in a carry from bit 7 of the accumulator?



**Figure 4-9**  
Condition code registers monitor the operations in the accumulator.

To keep track of these conditions, most microprocessors have a group of single bit registers called condition code registers. Three of these registers are shown in Figure 4-9. They are the Negative (N) Register, the Zero (Z) Register, and the Carry (C) Register.

**Negative (N) Register** Recall that negative numbers are expressed in two's complement form. Using this system, the most significant bit determines whether or not the number is negative. In an 8-bit byte, bit 7 is a 1 if the two's complement number is negative. Thus, the N register monitors bit 7 of the accumulator. Immediately after an operation that involves the accumulator, the N register looks at bit 7 to see if the number is negative. If so, the N register is set to 1. If the number in the accumulator is not negative, the N register is reset to 0.

Most operations that involve the accumulator affect the N register in this way — **but not all**. In a later unit we will point out how this register is affected by each instruction. In this unit, we will assume that the N register is affected as outlined above any time a number is added to, subtracted from, loaded into, or stored from the accumulator.

Another name for a condition code is a flag. Thus, the N register is sometimes called the N flag or the negative flag.

**Zero (Z) Register** This register monitors the accumulator looking for all zeros. Immediately after an operation that involves the accumulator, the zero-detect circuit looks at the resulting number. If all 8 bits are 0, the Z register is set to 1. Otherwise, the Z register is reset to 0. Most operations that involve the accumulator affect the Z register in this way.

**Carry (C) Register** The C register acts somewhat like an extension of the accumulator. You have seen that when two unsigned 8-bit numbers are added, the sum is frequently a 9-bit number. For example:

$$\begin{array}{r} \phantom{+} 1001 \ 0010 \quad \text{8-bit addend} \\ + \phantom{+} 1100 \ 0110 \quad \text{8-bit augend} \\ \hline 1 \ 0101 \ 1000 \quad \text{9-bit sum} \\ \text{carry} \uparrow \end{array}$$

Since the accumulator is an 8-bit register, the sum will not fit. The most significant bit (the carry) would be lost if you did not have another 1-bit register to hold it. This is the purpose of the C register. Any operation that causes a carry out of bit 7 will set the carry register to 1. Arithmetic operations that do not result in a carry will reset this register to 0.

The carry register is also used to keep track of "borrows" during subtract operations. If a subtraction requires a borrow for bit 7, the carry flag will also be set. For example, suppose you subtract an unsigned, binary number from a smaller unsigned binary number. The result will, of course, be a negative number. Moreover, bit 7 will have to "borrow" a bit to complete the subtraction. As a simple example, let's subtract 2 from 1. The subtraction looks like this

$$\begin{array}{r} \text{Borrow} \rightarrow 1 \\ \phantom{+} 0000 \ 0001 \quad \text{Minuend} \\ - \phantom{+} 0000 \ 0010 \quad \text{Subtrahend} \\ \hline 1111 \ 1111 \quad \text{Difference} \end{array}$$

The carry bit is set to 1 to indicate that a borrow operation occurred. Many subtraction operations do not require borrows. In these cases, the carry bit is reset to 0 to indicate that no borrow occurred.

Notice that the carry code can have different meanings, depending on the operation involved. That is, a 1 can mean either that a carry occurred or that a borrow occurred. The precise meaning of the 1 depends on whether the operation was an addition or a subtraction. We will discuss some additional aspects of the carry register in a later unit.

**Overflow (V) Register** The final condition code that is to be considered in this unit keeps track of two's complement overflow. Figure 4-10 shows how this register is connected in the MPU. A special circuit detects an overflow condition by monitoring bit 7 of the ALU's input and output lines. This circuit sets the V flag when an overflow occurs but clears it if no overflow occurs.

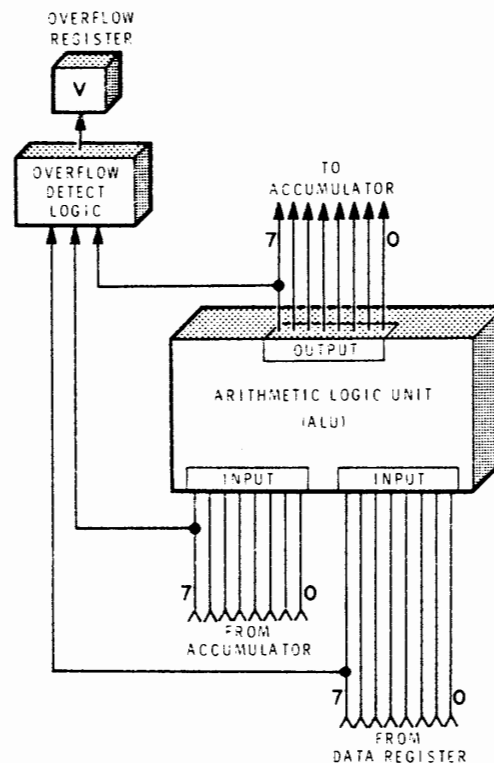


Figure 4-10

The overflow register monitors bit 7 of the ALU's input and output lines.

Let's see what is meant by two's complement overflow. Recall that the ALU adds numbers as if they were unsigned binary numbers. Even so, it can handle signed binary numbers if the proper bit patterns represent the negative numbers. This is the reason that the two's complement method of representing signed numbers has become so popular. A disadvantage of this system is that the magnitude of the number must be represented by 7 bits, since the eighth bit is used as the sign. Remember that a 1 in the MSB defines the number as negative.

Unfortunately, if two signed numbers are added and their sum exceeds 7-bits, the sign bit will be changed. For example, assume that a program adds  $+73_{10}$  and  $+96_{10}$ . The addition looks like this:

$$\begin{array}{r} \underline{0100\ 1001_2} \quad +73_{10} \\ \underline{0110\ 0000_2} \quad +96_{10} \\ 1010\ 1001_2 \quad 169_{10} \end{array}$$

The answer is correct if all the binary numbers represent unsigned quantities. However, using two's complement, the underlined bits represent sign bits. Therefore, the answer does **not** represent  $169_{10}$ . Instead, it represents  $-87_{10}$ . The reason for this error is that there was an overflow from bit 6 into the sign bit (bit 7). This is one of the situations that the V flag indicates.

When two's complement numbers having the same sign are added, the sum should have the same sign. That is, when two positive numbers are added, the sum should be positive. By the same token, when two negative numbers are added, the sum should be negative. However, an overflow can cause the sign to be reversed. The overflow logic detects this situation and sets the V flag whenever an overflow occurs.

The sign bit can also be upset during subtract operations. For example, when a negative number is subtracted from a positive number, the results should be positive. Remember that subtracting a negative number is tantamount to adding a positive number. However, in certain cases, an overflow can reverse the sign bit. This type of overflow occurs when the signs of the minuend and subtrahend are opposite and the difference has the sign of the subtrahend. This condition also sets the V flag.

## Conditional Branch Instructions

The conditional branch instructions available in our hypothetical microprocessor are shown in Figure 4-11. While these are largely self-explanatory, a couple of points should be mentioned.

INSTRUCTION	MNEMONIC	OPCODE	BRANCH IF
Branch If Carry Clear	BCC	24	C=0
Branch If Carry Set	BCS	25	C=1
Branch If Not Equal Zero	BNE	26	Z=0
Branch If Equal Zero	BEQ	27	Z=1
Branch If Plus	BPL	2A	N=0
Branch If Minus	BMI	2B	N=1
Branch If Overflow Clear	BVC	28	V=0
Branch If Overflow Set	BVS	29	V=1

Figure 4-11  
Conditional Branch Instructions.

The first instruction, Branch If Carry Clear (BCC), monitors the C register. If the carry register is reset to 0, the branch is implemented. Notice that the words “clear” and “reset” are used interchangeably in this regard. They both mean the register contains a 0.

The branch instructions that monitor the Z register can also be confusing. The Branch If Equal Zero (BEQ) instruction implements a branch when the Z register is set to 1. Recall that the Z register is set to 1 when the number in the accumulator is zero. Thus, you must remember that a 0 in the Z register means that the number in the accumulator is **not** zero.

These conditional branch instructions can be used with other instructions to make a wide range of decisions. They greatly increase the power of the microprocessor. More than any other type of instruction, the conditional branches are responsible for the MPU’s “intelligence.” In the next section, you will see how these instructions are used.

## Self-Test Review

10. What is the difference between an unconditional branch instruction and a conditional branch instruction?
11. What condition is tested by the branch if minus (BMI) instruction?
12. When is the N flag set?
13. When is the Z flag set?
14. During an add operation, the C flag is set. What does this represent?
15. During a subtract operation, the C flag is set. What does this indicate?
16. Often, when two positive 2's complement numbers are added, the sign bit of the answer will indicate a negative sum. This "error" can be spotted by checking which flag?
17. Under what condition will the BEQ instruction cause a branch to occur?
18. Under what condition will the BPL instruction cause a branch to occur?
19. When subtracting unsigned binary numbers, which flag indicates that the difference is a negative number?

## Answers

10. An unconditional branch instruction always causes a branch operation to occur. On the other hand, the conditional branch instruction implements a branch operation only if some specified condition is met.
11. The BMI instruction tests the Negative (N) register to see if it is set.
12. Generally speaking, the N flag is set if the previous instruction left a 1 in the MSB of the accumulator.
13. Generally, the Z flag is set if the previous instruction left all zeros in the accumulator.
14. During an add operation, the carry bit is set if there is a carry from bit 7 of the accumulator.
15. During a subtract operation, the carry bit is set if bit 7 had to "borrow" a bit to complete the subtraction.
16. This condition results from a two's complement overflow. Thus, the V flag will be set if this condition occurs.
17. The BEQ instruction causes a branch to occur only if the Z register is set.
18. The BPL instruction causes a branch to occur only if the N register is clear.
19. The carry flag.



## ALGORITHMS

An algorithm is a step-by-step procedure for doing a particular job. It generally involves doing a complex task by stringing together a series of simple steps. To illustrate the use of an algorithm, consider the following very simple example.

### Multiplying by Repeated Addition

Most microprocessors do not have hardware multiply capabilities. That is, they do not have a multiplication circuit nor a multiply instruction. Nevertheless, the microprocessor can be made to multiply by use of an algorithm. One procedure for doing this was discussed earlier. It involved adding the multiplicand to itself the number of times indicated by the multiplier. In the previous example, this was done by using a separate ADD instruction for each addition. This procedure is unsatisfactory for two reasons. First, it results in excessively long programs. Second, you must know the value of the multiplier so that you know how many ADD instructions to include.

A better approach, although still far from ideal, is to use a program loop that will multiply two numbers by repeated addition. For the time being, assume that the two numbers are both positive and that the product does not exceed  $255_{10}$ . Let's further assume that we use only the instructions which have been discussed up to this point. In fact, we will restrict ourselves to the instructions shown in Figure 4-12.

INSTRUCTION	MNEMONIC	ADDRESSING MODE			
		IMMEDIATE	DIRECT	RELATIVE	INHERENT
Load Accumulator	LDA	86	96		
Clear Accumulator	CLRA				4F
Decrement Accumulator	DECA				4A
Increment Accumulator	INCA				4C
Store Accumulator	STA		97		
Add	ADD	88	98		
Subtract	SUB	80	90		
Branch Always	BRA			20	
Branch if Carry Set	BCS			25	
Branch if Equal Zero	BEQ			27	
Branch if Minus	BMI			2B	
Halt	HLT				3E

Figure 4-12  
Instructions to be used.

The algorithm for multiplying by repeated addition is quite simple. To multiply A times B, you merely add A to a specific location B times. For example, to multiply 5 times 3, you clear a location and then add 5. You continue the addition until 5 has been added 3 times. The number in the affected location will then be  $15_{10}$  which is the product of 5 times 3.

The success of this operation depends on the microprocessor knowing when to stop. It must add 5 three times, but only three times. One way to keep track of the number of additions is to decrement the multiplier (3) each time an addition is made. When the multiplier reaches 0, the proper number of multiplications has been carried out.

Figure 4-13 is a flow chart that illustrates the algorithm. In the first two steps, the MPU clears the accumulator and stores the resulting number (0) in the product. This ensures that the product is zero before the first number is added. Next, it loads the multiplier and checks to see if the multiplier is 0. If so, the process is stopped since a multiplier of 0 dictates a product of 0.

In our example, the multiplier is 3; therefore, we exit the decision block via the "no" line. The next step tells us to decrement the multiplier. The new value of the multiplier (2) is stored for future use. Next, the product whose present value is 0 is loaded. Then, the multiplicand (5) is added so that the new value of the product becomes 5. This completes our first pass through the program. Remember that the multiplicand has been added once and that the multiplier has been reduced by one.

Notice that the program loops back to the input of the second block. The product which now has a value of 5 is stored back in memory. The multiplier (which is now 2) is loaded and tested. Because its value is not yet 0, the multiplier is decremented to 1 and stored again. The product (whose value is now 5) is then loaded and the multiplicand is added so that a new value of  $10_{10}$  is obtained.

The program loops again and the new product ( $10_{10}$ ) is stored. The multiplier (whose value is now 1) is loaded and tested. Because its value is still not 0, it is decremented again. Notice that the value of the multiplier is now 0. This value is stored away, the product ( $10_{10}$ ) is fetched, and the multiplicand is added once more. The new value of the product becomes  $15_{10}$ .

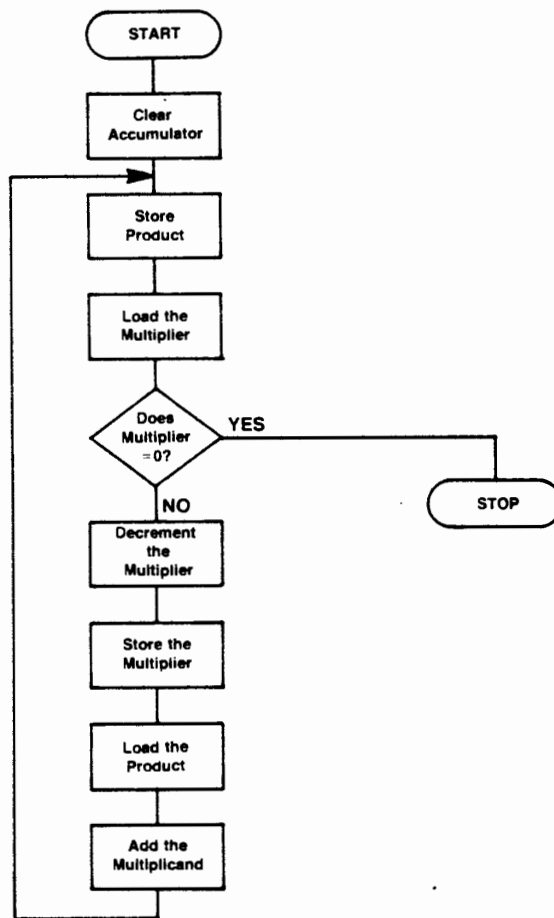


Figure 4-13  
Flow chart for multiplying by repeated  
addition.

The program loops again and the product is stored. The multiplier is loaded and tested. Recall that the value of the multiplier is now 0. Consequently, we exit the decision block via the "yes" line. The program has accomplished its task and it now stops. Notice that the value of the product is  $15_{10}$  which is the proper answer for  $5 \times 3$ .

The next task is to convert the flow chart to a program that the computer can execute. Figure 4-14 shows such a program. Carefully compare this program to the flow chart paying particular attention to the comments column. Work through the program on paper and verify that it will multiply the numbers at addresses  $11_{16}$  and  $12_{16}$ . Although 3 and 5 are used in this example, the program will work for any values of multiplier and multiplicand as long as the product does not exceed  $255_{10}$ .

HEX ADDRESS	HEX CONTENTS	MNEMONIC/HEX CONTENTS	COMMENTS
00	4F	CLRA	Clear the accumulator.
01	97	STA	Store the product
02	13	13	in location $13_{16}$ .
03	96	LDA	Load the accumulator with the
04	12	12	multiplier from location $12_{16}$ .
05	27	BEQ	If the multiplier is equal to zero,
06	09	09	branch down to the Halt instruction.
07	4A	DECA	Otherwise, decrement the multiplier.
08	97	STA	Store the new value of the
09	12	12	multiplier back in location $12_{16}$ .
0A	96	LDA	Load the accumulator with the
0B	13	13	product from location $13_{16}$ .
0C	9B	ADD	Add
0D	11	11	the multiplicand to the product.
0E	20	BRA	Branch back to instruction
0F	F1	F1	in location 01.
10	3E	HLT	Halt.
11	05	05	Multiplicand.
12	03	03	Multiplier.
13	—	—	Product.

Figure 4-14

This program multiplies the numbers at addresses  $11_{16}$  and  $12_{16}$ , and places their product at address  $13_{16}$ .

## Dividing by Repeated Subtraction

Another interesting algorithm is one that allows the microprocessor to divide by repeated subtraction. The technique is to keep track of the number of times that the divisor can be subtracted from the dividend. For example, suppose you wish to divide  $47_{10}$  by  $15_{10}$ . The divisor can be subtracted 3 times:

First subtraction      Second Subtraction      Third Subtraction

$$\begin{array}{r} 47_{10} \\ -15_{10} \\ \hline 32_{10} \end{array} \rightarrow \begin{array}{r} 32_{10} \\ -15_{10} \\ \hline 17_{10} \end{array} \rightarrow \begin{array}{r} 17_{10} \\ -15_{10} \\ \hline 2_{10} \end{array}$$

Because three subtractions occurred, the quotient is 3. Also, because 2 was left after the last subtraction, the remainder is 2. We can verify this by long division:

$$\begin{array}{r} \text{divisor} \quad \rightarrow \quad 15_{10} \overline{) 47_{10}} \\ \underline{\cdot 45_{10}} \\ 2_{10} \end{array} \quad \begin{array}{l} \leftarrow \text{Quotient} \\ \leftarrow \text{Dividend} \\ \leftarrow \text{Remainder} \end{array}$$

The microprocessor keeps track of the number of subtractions by incrementing the quotient by one each time a subtraction occurs. Of course, the quotient must be initially set to zero.

The divisor is subtracted from the dividend until any further subtraction would result in a negative number. The MPU can use the BMI instruction to check for a negative result on each loop. The negative result is the indication that the process is finished.

A flow chart for this algorithm is shown in Figure 4-15. The actual program is shown in Figure 4-16. The program is arbitrarily placed in locations 00 through 10<sub>16</sub>. The dividend (47<sub>10</sub>) is at address 11<sub>16</sub> while the divisor (15<sub>10</sub>) is at address 12<sub>16</sub>. When executed, the program will produce the quotient at location 13<sub>16</sub> and the remainder at location 11<sub>16</sub>.

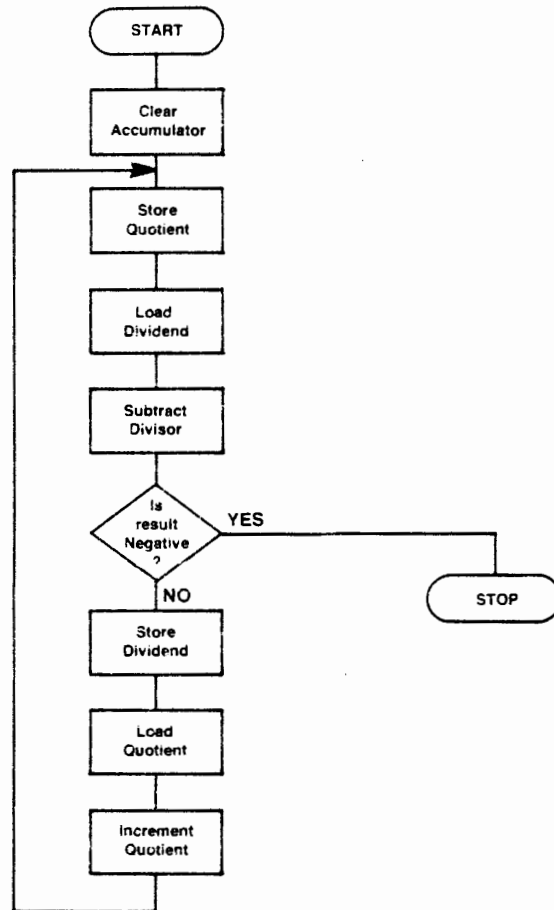


Figure 4-15  
Flow chart for dividing by repeated subtraction.

HEX ADDRESS	HEX CONTENTS	MNEMONIC/HEX CONTENTS	COMMENTS
00	4F	CLRA	Clear the accumulator.
01	97	STA	Store in the quotient which
02	13	13	is at address location 13 <sub>16</sub> .
03	96	LDA	Load the accumulator with the
04	11	11	dividend from location 11 <sub>16</sub> .
05	9C	SUB	Subtract the
06	12	12	divisor from the dividend.
07	2B	BMI	If the difference is negative, branch
08	07	07	down to the Halt instruction.
09	97	STA	Otherwise, store the difference
0A	11	11	back in location 11 <sub>16</sub> .
0B	96	LDA	Load the accumulator with the
0C	13	13	quotient.
0D	4C	INCA	Increment the quotient by one.
0E	20	BRA	Branch back to instruction
0F	F1	F1	in location 01.
10	3E	HLT	Halt.
11	2F	2F	Dividend (47 <sub>10</sub> ).
12	0F	0F	Divisor (15 <sub>10</sub> ).
13	—	—	Quotient.



Figure 4-16

This program divides by repeatedly subtracting the divisor from the dividend.

Refer to the flow chart and the comments column of the program. Before reading further, try running through the program on paper. This will give you a feel for how the computer solves the problem.

Now let's go through the program to see what it does. The first two instructions clear the quotient. Next, the dividend (47<sub>10</sub>) is loaded into the accumulator and the divisor (15<sub>10</sub>) is subtracted. The BMI instruction is used to examine the difference (32<sub>10</sub>). Since the difference is not minus, the branch does not occur. Consequently, the next instruction stores the difference (32<sub>10</sub>) back in the location from which the dividend came. In effect, the difference becomes the new dividend. Next the quotient (0) is loaded and is incremented to 1. The program then branches back to the instruction in location 01. This instruction stores the quotient (1) back in location 13<sub>16</sub>.

On the next pass through the program, the new dividend ( $32_{10}$ ) is loaded and the divisor ( $15_{10}$ ) is subtracted again. This produces a difference of ( $17_{10}$ ). Since the difference is not negative, the BMI instruction does not cause a branch. Thus, the difference is stored back in location  $11_{16}$ . The quotient is loaded into the accumulator and is incremented to 2. The BRA instruction causes the program to loop once again. The STA instruction in location 01 stores the quotient (2) back in location  $13_{16}$ .

On the third pass the dividend ( $17_{10}$ ) is loaded and the divisor ( $15_{10}$ ) is subtracted a third time. The difference (02) is still not negative so no branch occurs. The difference is stored away; the quotient is loaded and is incremented to 03. Notice that this is the proper final value for the quotient. Therefore, on the next pass, the MPU should be able to break out of the loop.

The quotient is stored back in location  $13_{16}$ . The dividend, which now has a value of 2, is loaded. The divisor ( $15_{10}$ ) is subtracted, leaving a negative number ( $-13_{10}$ ) in the accumulator. The BMI instruction recognizes that this is a negative number and implements a branch operation. Notice that the MPU branches forward to the HLT instruction. Thus, the program ends with the quotient set to 3. The remainder is at address  $11_{16}$ . That is, the remainder is what remains of the dividend after the third subtraction.

It may bother you that there were four subtractions and that a negative difference resulted from the last subtraction. However, you will recall that the quotient was incremented only on the first three of these subtractions. Thus, the final quotient is 3. Moreover, the negative difference that resulted during the last subtraction was never stored. Consequently, the remainder was 2 when the program ended.

This program does have some drawbacks. For one thing, neither the dividend nor the divisor can exceed  $127_{10}$ . Also, only positive numbers can be used. Finally, the program gets hung up in an endless loop if the initial value of the divisor is zero. While division by zero is not allowed in mathematics, some provision would be made in a practical program to recognize this eventuality. Since the program is for demonstration purposes, we will live with these shortcomings for the time being.



## Converting BCD to Binary

When a microprocessor is used with a terminal such as a teletypewriter, numerals are entered as ASCII characters. For example, the number  $237_{10}$  is entered into memory as three ASCII characters:

Numeral	ASCII Character
2	0011 0010
3	0011 0011
7	0011 0111

Notice that the four least significant bits of the ASCII character represent the BCD value of the corresponding numeral. Thus, we can convert these ASCII characters to BCD numbers simply by eliminating the four most significant bits. This technique was demonstrated in an earlier experiment.

While the microprocessor does have some BCD capability, it is often desirable to convert BCD numbers to binary. The technique for doing this illustrates another useful algorithm.

The BCD representation for  $237_{10}$  is:

0010 ← hundreds BCD digit  
 0011 ← tens BCD digit  
 0111 ← units BCD digit

Notice that in this example 0010 represents two hundred, 0011 represents thirty, and 0111 represents seven. Because of this, there is a simple procedure for converting BCD to binary. Starting with an initial value of zero, the MPU adds  $100_{10}$  as many times as indicated by the hundreds digit. It then adds  $10_{10}$  as indicated by the tens digit. Finally, the value of the units digit is added on to the result. The steps involved look like this:

1100100 <sub>2</sub>	100 <sub>10</sub>	}	One hundred added
1100100 <sub>2</sub>	100 <sub>10</sub>		2 times
1010 <sub>2</sub>	10 <sub>10</sub>	}	Ten added three times
1010 <sub>2</sub>	10 <sub>10</sub>		
1010 <sub>2</sub>	10 <sub>10</sub>		
0111 <sub>2</sub>	7 <sub>10</sub>		7 units added
<hr/> 11101101 <sub>2</sub> = 237 <sub>10</sub>			

As you can see, this procedure ends with a binary result of 1110 1101, which is the binary representation for  $237_{10}$ .

A flow chart for this procedure is shown in Figure 4-17. Here, the first step is to clear the binary result. We will be adding to this result, so it must start out at zero.

Next the program enters a loop in which it adds  $100_{10}$  to the binary result the number of times indicated by the hundreds digit of the BCD number. The hundreds digit is loaded and tested for zero. If it is not zero, the hundreds digit is decremented and stored back in memory. Then the binary result is loaded and  $100_{10}$  is added. The result is stored away and the loop is repeated. In our example, the hundreds digit was initially 2. Thus, this loop is repeated twice. The binary result will have the value  $1100\ 1000_2$  ( $200_{10}$ ) when the hundreds digit is reduced to zero. At that time, the program exits the decision block via the "yes" line and immediately encounters a second loop.

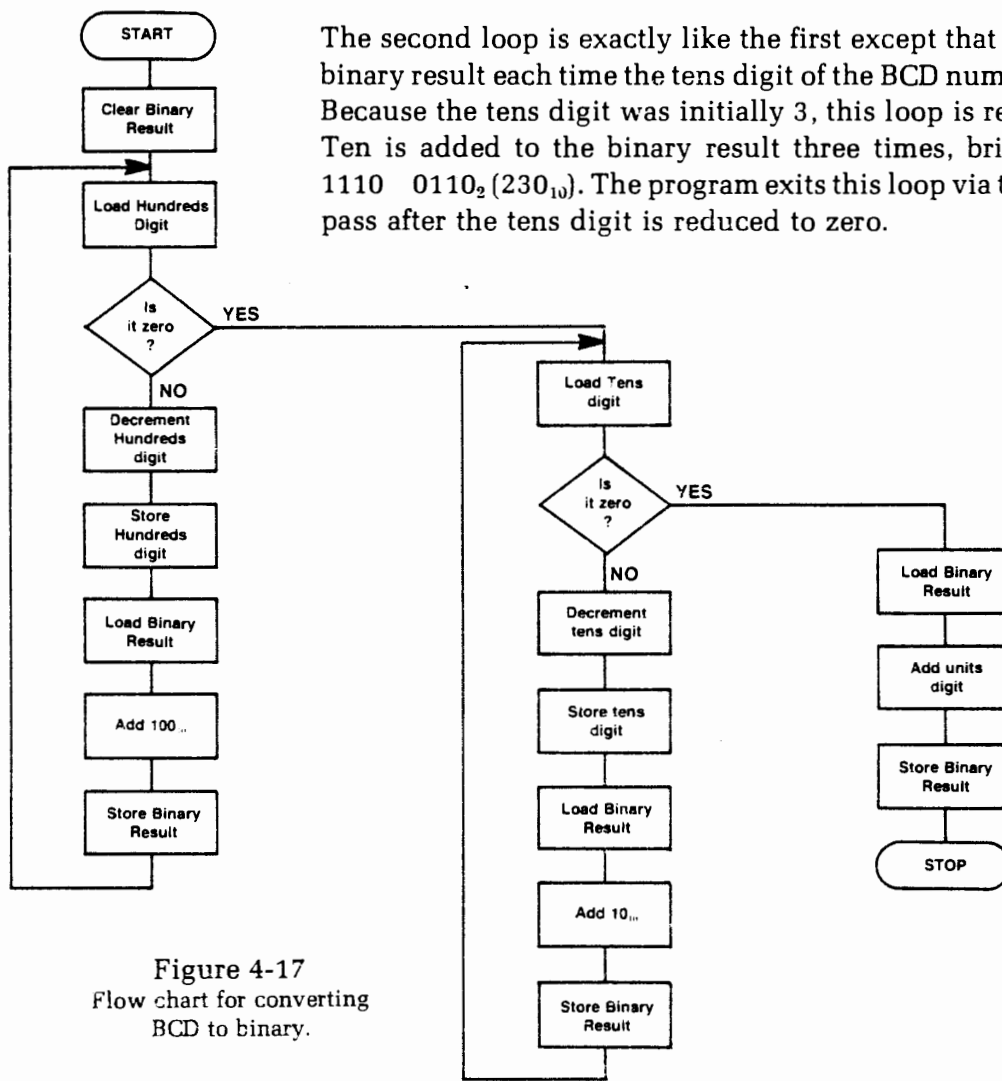


Figure 4-17  
Flow chart for converting  
BCD to binary.

The final three blocks add the units digit to the binary result. In our example, the units digit was  $7_{10}$ . This brings the final binary result to  $1110\ 1101_2$ . Notice that this is the proper binary representation for the unsigned number  $237_{10}$ .

A program that carries out this operation is shown in Figure 4-18. The three digit BCD number is stored in locations  $28_{16}$ ,  $29_{16}$ , and  $2A_{16}$ . The binary equivalent will be computed and placed in location  $2B_{16}$ . Before reading further, try to work through the program. Refer to the flow chart and the comments column as you trace out the sequence that the MPU will follow.

HEX ADDRESS	HEX CONTENTS	MNEMONIC/HEX CONTENTS	COMMENTS
00	4F	CLRA	Clear the accumulator.
01	97	STA	Store 00
02	2B	2B	in location 2B. This clears the binary result.
03	96	LDA	Load direct
04	28	28	the hundreds BCD digit.
05	27	BEQ	If the hundreds digit is zero, branch
06	0B	0B	forward to the instruction in location 12...
07	4A	DECA	Otherwise, decrement the accumulator.
08	97	STA	Store the result as the new
09	28	28	hundreds BCD digit.
0A	96	LDA	Load direct
0B	2B	2B	the binary result.
0C	8B	ADD	Add immediate
0D	64	64	100... to the binary result.
0E	97	STA	Store away the new
0F	2B	2B	binary result.
10	20	BRA	Branch
11	F1	F1	back to the instruction in location 03...
12	96	LDA	Load direct
13	29	29	the tens BCD digit.
14	27	BEQ	If the tens BCD digit is zero, branch
15	0B	0B	forward to the instruction in location 21...
16	4A	DECA	Otherwise, decrement the accumulator.
17	97	STA	Store the result as the new
18	29	29	tens BCD digit.
19	96	LDA	Load direct
1A	2B	2B	the binary result.
1B	8B	ADD	Add immediate
1C	0A	0A	10... to the binary result.
1D	97	STA	Store away the new
1E	2B	2B	binary result.
1F	20	BRA	Branch
20	F1	F1	back to the instruction in location 12...
21	96	LDA	Load direct
22	2B	2B	the binary result.
23	9B	ADD	Add direct
24	2A	2A	the units BCD digit.
25	97	STA	Store away the new
26	2B	2B	binary result.
27	3E	HLT	Halt.
28	02	02	Hundreds BCD digit.
29	03	03	Tens BCD digit.
2A	07	07	Unit BCD digit.
2B	—	—	Reserved for the binary result.

Figure 4-18  
 Program for converting BCD to binary.

Now let's briefly go through the program. The first two instructions clear the location at which the binary number will be formed.

Next, the program enters the first loop, which is shown as the first shaded area. In this loop, the hundreds digit is loaded and tested for zero. If not zero, it is decremented and stored away. Then the binary result is loaded and  $100_{10}$  is added. The result is stored away and the loop is repeated. Because the hundreds digit was 02 initially,  $100_{10}$  will be added to the binary result twice. Thus, upon leaving this loop, the binary result will have the value  $200_{10}$ . The MPU escapes this loop when the BEQ instruction at address 05 detects that the hundreds digit has been reduced to zero. The branch is to the second loop which is shown as an unshaded area.

In the second loop, the tens digit is loaded and tested for zero. If not zero, it is decremented and stored away. Then the binary number is loaded,  $10_{10}$  is added, and the result is stored away. This loop is repeated until the tens digit is reduced to zero. Because the tens digit was initially three, the loop is repeated three times so that thirty is added to the binary number. The BEQ instruction at address  $14_{16}$  allows the MPU to escape the loop and branch to the final program segment.

This final segment is the last shaded area. Here, the binary result is loaded and the units digit is added. This brings the binary result to  $237_{10}$ . Then the result is stored and the program halts. While the number  $237_{10}$  was used in this example, the program will convert any BCD number between 000 and  $255_{10}$  to its binary equivalent.

## Converting Binary to BCD

A microprocessor generally manipulates data in the form of straight binary numbers. However, before the results can be transmitted to the outside world, the data is often converted back to BCD. Frequently, this is an intermediate step in converting back to ASCII.

The binary-to-BCD conversion is the reverse of the process that occurred in the previous program. The MPU must determine how many times  $100_{10}$  can be subtracted from the binary number. The answer becomes the hundreds BCD digit. After the  $100_{10}$  has been subtracted as many times as possible,  $10_{10}$  is subtracted repeatedly from the remaining number. The number of subtractions becomes the tens BCD digit. Finally, after  $10_{10}$  has been subtracted as many times as possible, the remaining number becomes the units BCD digit.

For the number  $1110\ 1101_2$  ( $237_{10}$ ), the process looks likes this:

1110 1101	237	
-0110 0100	-100	
1000 1001	137	← hundreds digit = 2
-0110 0100	-100	
0010 0101	37	
-0000 1010	-10	
0001 1011	27	← tens digit = 3
-0000 1010	-10	
0001 0001	17	
-0000 1010	-10	
0000 0111	7	← units digit = 7

One hundred can be subtracted twice. Thus, the hundreds digit is  $2_{10}$  or  $0010_2$ . From the remainder, ten can be subtracted three times. Thus, the tens digit is  $3_{10}$  or  $0011_2$ . Finally, the remainder of  $7_{10}$  or  $0111_2$  becomes the units digit. The BCD representation is  $0010\ 0011\ 0111$ .

Figure 4-19 shows the flow chart for this procedure. The first three blocks clear the hundreds, tens, and units digits of the BCD result. Then the binary number that is to be converted to BCD is loaded and  $100_{10}$  is subtracted. The outcome is tested to see if a negative number resulted. If not, the result is stored away. The hundreds digit is loaded, incremented, and stored away. The loop is repeated until  $100_{10}$  can no longer be subtracted. In our example,  $100_{10}$  can be subtracted twice. Therefore, the hundreds digit is incremented to 2. The third subtraction of  $100_{10}$  gives a negative result. This allows the MPU to escape the first loop.

The second loop increments the tens digit to the proper value by subtracting  $10_{10}$  repeatedly and keeping track of the number of subtractions. In our example, this loop is repeated three times. Consequently, the tens digit is incremented to 3. The binary number that is left over after  $10_{10}$  is subtracted the proper number of times becomes the units digit. That is, upon escaping the second loop, the remaining binary number is stored in the units digit. In our example, the remaining number, and therefore the units digit, is 7.

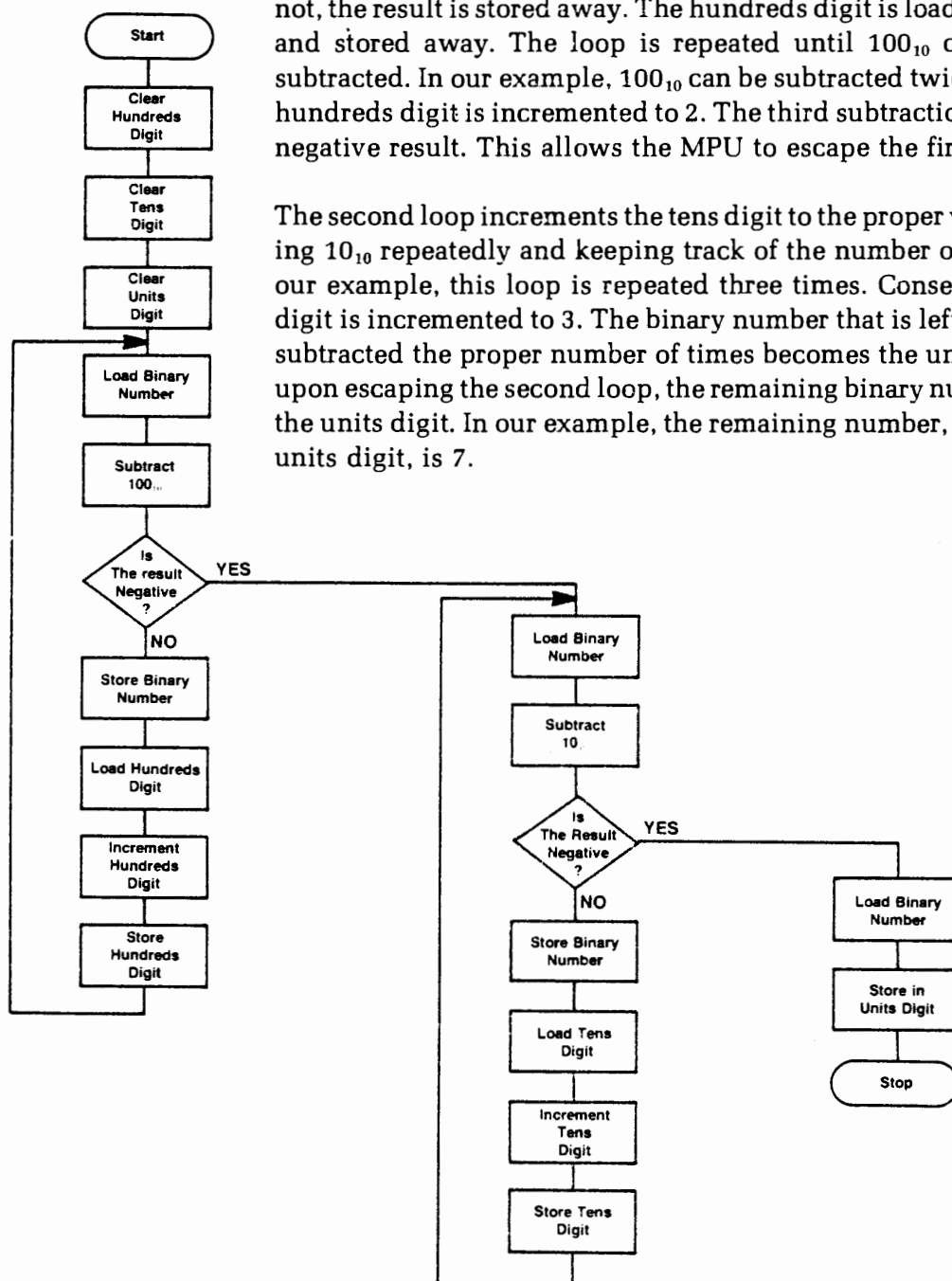


Figure 4-19

Flow chart for converting a binary number to a BCD number.

The program that carries out this procedure is shown in Figure 4-20. At this point, you should be able to interpret the program from the comments given. However, a couple of points should be explained briefly. Any unsigned binary number from 0000 0000 to 1111 1111 can be placed at address 2A<sub>16</sub>. The computer will convert this number into its BCD equivalent. The hundreds digit will appear at address 2B<sub>16</sub>, the tens digit at 2C<sub>16</sub>, and the units digit at 2D<sub>16</sub>. The decision making instructions at addresses 0B<sub>16</sub> and 1A<sub>16</sub> are Branch if Carry Set (BCS) instructions. Because these instructions follow immediately after SUB instructions, the carry flag will indicate whether or not a borrow occurred. In effect, the BCS instructions decide: "Was the result of the subtraction a negative number?"

HEX ADDRESS	HEX CONTENTS	MNEMONIC/HEX CONTENTS	COMMENTS
00	4F	CLRA	Clear the accumulator.
01	97	STA	Store 00
02	2B	2B	in location 2B <sub>16</sub> . This clears the hundreds digit.
03	97	STA	Store 00
04	2C	2C	in location 2C <sub>16</sub> . This clears the tens digit.
05	97	STA	Store 00
06	2D	2D	in location 2D <sub>16</sub> . This clears the units digit.
07	96	LDA	Load direct
08	2A	2A	the binary number to be converted.
09	80	SUB	Subtract immediate
0A	64	64	100 <sub>10</sub> .
0B	25	BCS	If a borrow occurred, branch
0C	09	09	forward to the instruction in location 16 <sub>16</sub> .
0D	97	STA	Otherwise, store the result of the subtraction
0E	2A	2A	as the new binary number.
0F	96	LDA	Load direct
10	2B	2B	the hundreds digit of the BCD result.
11	4C	INCA	Increment the hundreds digit.
12	97	STA	Store the hundreds digit
13	2B	2B	back where it came from.
14	20	BRA	Branch
15	F1	F1	back to the instruction at address 07 <sub>16</sub> .
16	96	LDA	Load direct
17	2A	2A	the binary number.
18	80	SUB	Subtract immediate
19	0A	0A	10 <sub>10</sub> .
1A	25	BCS	If a borrow occurred, branch
1B	09	09	forward to the instruction in location 25 <sub>16</sub> .
1C	97	STA	Otherwise, store the result of the subtraction
1D	2A	2A	as the new binary number.
1E	96	LDA	Load direct
1F	2C	2C	the tens digit.
20	4C	INCA	Increment the tens digit.
21	97	STA	Store the tens digit
22	2C	2C	back where it came from.
23	20	BRA	Branch
24	F1	F1	back to the instruction at address 16 <sub>16</sub> .
25	96	LDA	Load direct
26	2A	2A	the binary number.
27	97	STA	Store it in
28	2D	2D	the units digit.
29	3E	HLT	Halt.
2A	—	—	Place binary number to be converted at this address.
2B	—	—	Hundreds digit
2C	—	—	Tens digit
2D	—	—	Units digit
			} Reserved for BCD result.

Figure 4-20  
 Program for converting a binary  
 number to a BCD number.

## Self-Test Review

20. What is an algorithm?
21. What type of instruction is used to make a decision?
22. Refer to the program in Figure 4-14. If the multiplier is  $8_{16}$  and the multiplicand is  $15_{16}$ , how many times will the BEQ instruction be executed?
23. Refer to the program in Figure 4-16. What is the largest number that can be used as a dividend?
24. How could this program be modified so it could handle unsigned dividends up to  $255_{10}$ ?
25. When this program halts, where will the remainder be located?
26. Refer to the program in Figure 4-18. Assume that addresses  $28_{16}$ ,  $29_{16}$ , and  $2A_{16}$  contain 01, 09, and 08 respectively. How many times will  $100_{10}$  be added to address  $2B_{16}$ ?
27. How many times will  $10_{10}$  be added?
28. Refer to the program in Figure 4-20. What is the purpose of the first four instructions?
29. What is the largest binary number that this program can convert to BCD?



## Answers

20. An algorithm is a step-by-step procedure for doing a particular job.
21. Conditional branch instruction.
22. Nine times.
23.  $+127_{10}$  or  $0111\ 1111_2$ .
24. Change the BMI instruction to BCS.
25. At address  $11_{16}$ .
26. Once.
27. Nine times.
28. The first four instructions clear the locations where the BCD equivalent will be stored.
29.  $1111\ 1111_2$  or  $255_{10}$ .

## ADDITIONAL INSTRUCTIONS

Before leaving this unit, you should also look at four additional instructions. The names, opcodes and mnemonics of these instructions are shown in Figure 4-21.

NAME	MNEMONIC	HEX OPCODE		
		IMMEDIATE	DIRECT	INHERENT
ADD WITH CARRY	ADC	89	99	
SUBTRACT WITH CARRY	SBC	82	92	
ARITHMETIC SHIFT ACCUMULATOR LEFT	ASLA			48
DECIMAL ADJUST ACCUMULATOR	DAA			19

Figure 4-21  
Four new instructions.

Recall that the ALU always adds numbers as if they were unsigned binary numbers. When it adds 8-bit numbers, a carry often occurs from the MSB, setting the C flag. Thus, you can think of the carry flag as an extension of the accumulator. Let's look at some instructions that use the carry flag.

### Add With Carry (ADC) Instruction

This instruction is similar to the ADD instruction discussed earlier with one important difference. If the carry bit is set to 1 before this instruction is executed, 1 is added to the LSB of the sum. However, if the carry bit is 0 prior to execution, then no carry is added. The effect is the same as having the carry bit from the previous operation added to the result of the present operation.

Like the ADD instruction, the ADC instruction has two addressing modes: immediate and direct. As shown in Figure 4-21, the opcode for "ADD With Carry Immediate" is  $89_{16}$ , while the opcode for "Add With Carry Direct" is  $99_{16}$ .

A primary use of the ADC instruction is to simplify **multiple-precision arithmetic**. Multiple-precision means that two or more bytes are used to represent a number. Recall that a single byte can represent unsigned binary numbers with values up to  $255_{10}$ . However, much larger numbers can be represented by using two or more bytes. Two bytes (16 bits) can represent unsigned binary values up to  $2^{16} - 1$  or  $65,535_{10}$ . Three bytes can represent values to  $16,777,215_{10}$ ; etc. Thus, the MPU can handle numbers of virtually any size simply by stringing the proper number of bytes together.

Suppose, for example, that two very large numbers are to be added. Figure 4-22 shows how the addition might look on paper. Notice that two 24-bit numbers are being added to form a 24-bit sum. The MPU is restricted to operating on data in 8-bit bytes. Thus, each quantity involved must be represented by three bytes.

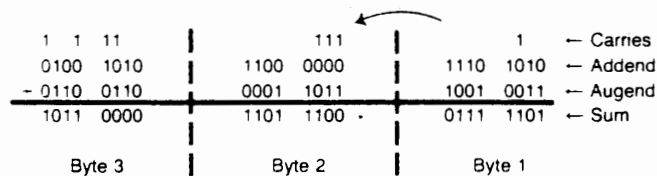


Figure 4-22  
Multiple-precision addition.

The MPU must be instructed to add the first byte of the addend to the first byte of the augend. This forms the first byte of the sum. Next the MPU must add the second bytes of the addend and augend. However, you will notice that there was a carry from the first byte to the second byte. If this carry is not added with the second bytes, the sum will be in error. The ADC instruction performs this operation automatically.

The program for adding the multiple-byte numbers could be written as shown in Figure 4-23. The three byte addend is stored in locations 13<sub>16</sub> through 15<sub>16</sub> while the augend is stored in locations 16<sub>16</sub> through 18<sub>16</sub>. Verify that the hexadecimal contents shown are the same as the binary values given in Figure 4-22.

HEX ADDRESS	HEX CONTENTS	MNEMONIC/HEX CONTENTS	COMMENTS
00	36	LDA	Load accumulator direct with
01	13	13	least significant byte of addend.
02	9B	ADD	Add direct
03	16	16	least significant byte of augend.
04	97	STA	Store result in
05	19	19	least significant byte of sum.
06	96	LDA	Load accumulator direct with
07	14	14	next byte of addend.
08	99	ADC	Add with carry direct
09	17	17	next byte of augend.
0A	97	STA	Store result in
0B	1A	1A	next byte of sum.
0C	96	LDA	Load accumulator direct with
0D	15	15	most significant byte of addend.
0E	99	ADC	Add with carry direct
0F	18	18	most significant byte of augend.
10	97	STA	Store result in
11	1B	1B	most significant byte of sum.
12	3E	HLT	Halt.
13	EA	EA	Least significant byte
14	C0	C0	Addend.
15	4A	4A	
16	93	93	Least significant byte
17	1B	1B	Augend
18	66	66	
19	—	—	Most significant byte
1A	—	—	Reserved for sum.
1B	—	—	

Page 4-48

Reserved  
for sum.

Figure 4-23  
Program for multiple-precision  
addition.

The first two instructions add the least significant bytes of the addend and augend. The ADD instruction is used because the MPU need not consider earlier carries. The first byte of the resulting sum is stored in location 19<sub>16</sub>.

The next two instructions add the next two bytes. This time the ADC instruction is used because the MPU must consider the carry from the previous addition. The second byte of the sum is placed in location 1A<sub>16</sub>.

Finally, the last two bytes are added using the ADC instruction. The final byte of the sum is stored in location 1B<sub>16</sub>. The program halts when the addition is completed.

## Subtract With Carry (SBC) Instruction

This instruction simplifies multiple-precision subtraction. You will recall that during subtract operations the carry flag indicates whether or not a borrow operation occurred. For this reason, this instruction can be thought of as a subtract with borrow operation.

The SBC instruction subtracts the subtrahend from the minuend just as the SUB instruction did. However, the SBC instruction has an additional step in that the carry bit is also subtracted. As with the other add and subtract instructions, both immediate and direct addressing modes are possible. The opcodes for both modes are shown in Figure 4-21.

Figure 4-24 illustrates how multiple-precision numbers can be subtracted. Notice that, during the course of this subtraction, byte 1 must "borrow" a 1 from byte 2. The SBC instruction allows the MPU to do this.

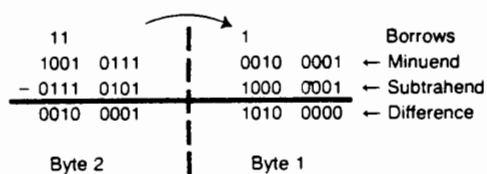


Figure 4-24  
Multiple-precision subtraction.

HEX ADDRESS	HEX CONTENTS	MNEMONIC/HEX CONTENTS	COMMENTS
00	96	LDA	Load accumulator direct with
01	0D	0D	least significant byte of minuend.
02	90	SUB	Subtract direct
03	0F	0F	least significant byte of subtrahend.
04	97	STA	Store result in
05	11	11	least significant byte of difference.
06	96	LDA	Load accumulator direct with
07	0E	0E	most significant byte of minuend.
08	92	SBC	Subtract with carry
09	10	10	most significant byte of the subtrahend.
0A	97	STA	Store result in
0B	12	12	most significant byte of the difference.
0C	3E	HLT	Halt
0D	21	21	Least significant byte } Minuend.
0E	97	97	Most significant byte }
0F	81	81	Least significant byte } Subtrahend.
10	75	75	Most significant byte }
11	—	—	Least significant byte } Difference.
12	—	—	Most significant byte }

Figure 4-25  
Program for multiple-precision  
subtraction.

Figure 4-25 shows a simple program for performing the subtraction. The double-precision minuend is at addresses  $0D_{16}$  and  $0E_{16}$ , while the subtrahend is at addresses  $0F_{16}$  and  $10_{16}$ . The program computes the difference and stores it in locations  $11_{16}$  and  $12_{16}$ .

The first instruction loads the least significant byte of the minuend. Next, the corresponding byte of the subtrahend is subtracted. Since the subtrahend byte is larger, a borrow is indicated. Consequently, the carry flag is set to 1. Notice that the SUB rather than the SBC instruction is used. This is done because the first byte should not be affected by any previous borrow or carry. The result of the subtraction is stored away to become the least significant byte of the difference.

The most significant byte of the minuend is loaded next and the corresponding byte of the subtrahend is subtracted. However, this time the SBC instruction is used. And since the carry flag is set, an additional 1 is subtracted from the minuend to complete the borrow operation. The result of the subtraction becomes the most significant byte of the difference.

## Arithmetic Shift Accumulator Left (ASLA) Instruction

The ASLA instruction shifts the contents of the accumulator to the left by one space. Figure 4-26 illustrates the repeated execution of this instruction. Figure 4-26A shows the condition of the accumulator and carry bit. In this example, the number in the accumulator is arbitrarily assumed to be  $10_{10}$ . Also, the carry bit is arbitrarily assumed to be cleared.

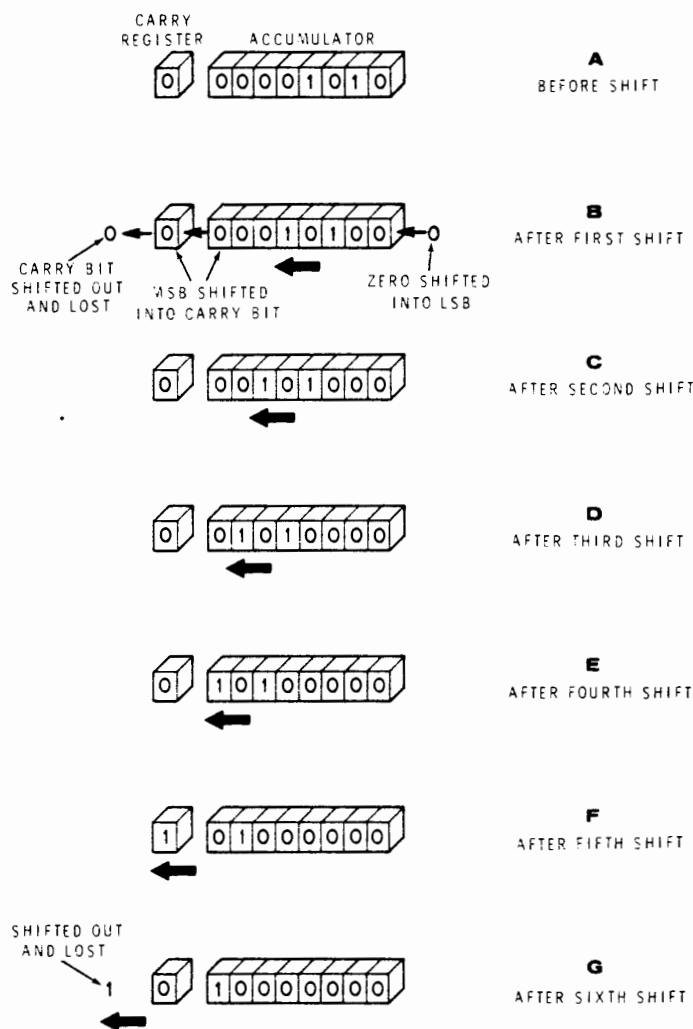


Figure 4-26  
Repeatedly implementing the ASLA  
instruction.

Figure 4-26B shows the contents of the accumulator and carry bit after the ASLA instruction is executed. Notice that the number is shifted one bit to the left. Also, a 0 is shifted into the LSB. At the same time, the MSB is shifted into the carry bit. The old carry bit is shifted out and is lost.

You can understand one purpose of this instruction by examining the numbers in the accumulator before and after the instruction is executed. Before the shift, the number is  $10_{10}$ ; afterwards the number is  $20_{10}$ . The number has been doubled. If you will try several different examples, you will see that any binary number can be multiplied by two simply by shifting the number one bit to the left. This holds true as long as the capacity of the accumulator is not exceeded.

Figures 4-26C through G show what happens if the MPU continues to execute ASLA instructions. The number continues to double. The number in the accumulator becomes  $40_{10}$ , then  $80_{10}$ , then  $160_{10}$ . Each shift multiplies the number by two. On the fifth shift, the capacity of the accumulator is exceeded as the most significant 1 bit shifts into the carry bit. After the sixth shift, the leading 1 is lost altogether. When you use this technique to multiply by two or by a power of two, you must not exceed the capacity of the accumulator.

Another use of the ASLA instruction is to pack two BCD digits in a single byte. Earlier when we worked with BCD numbers, we assumed that each BCD digit resided in a separate memory byte. However, because a BCD digit has only 4 bits, memory space is wasted by assigning each digit a separate byte. Frequently, it is more desirable to "pack" two BCD digits into a single byte. A simple routine for doing this is shown in Figure 4-27. If dozens of BCD numbers are to be manipulated, a routine that uses a procedure similar to this can save substantial memory space. At the same time, it puts the BCD numbers into a more convenient and usable form.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/HEX CONTENTS	COMMENTS
00	96	LDA	Load into the accumulator direct the unpacked most significant BCD digit.  } Shift it four places to the left.
01	0C	0C	
02	48	ASLA	
03	48	ASLA	
04	48	ASLA	
05	48	ASLA	Add the unpacked least significant BCD digit. Store the result as two packed BCD digits. Halt Packed BCD digits. Most significant BCD digit (unpacked). Least significant BCD digit (unpacked).
06	9B	ADD	
07	0D	0D	
08	97	STA	
09	0B	0B	
0A	3E	HLT	
0B	—	—	
0C	—	—	
0D	—	—	

Figure 4-27  
 Program for packing two  
 BCD digits into a single byte.



The procedure carried out by the program is quite simple. The most significant BCD digit is loaded into the accumulator. It is then shifted four places to the left to make room for the least significant BCD digit. The least significant digit is then added to form a packed BCD number. The resulting single byte number is stored back in memory.

## Decimal Adjust Accumulator (DAA) Instruction

Earlier in this unit, the problems of converting from BCD to binary and back again were considered. While this conversion is frequently necessary, many microprocessors have some limited BCD arithmetic capabilities. Our hypothetical MPU has an instruction that greatly simplifies BCD arithmetic. It is called the Decimal Adjust Accumulator (DAA) Instruction. When used in conjunction with the ADD or ADC instruction, it allows the MPU to add BCD numbers directly without an intermediate binary conversion.

Recall that the ALU adds input data bytes as if they were unsigned binary numbers. Therefore, if two BCD digits are added, the sum may be incorrect. For example, assume that the MPU adds the BCD digits 0111 and 0101. The ALU produces the result

$$\begin{array}{r} 1 \\ 0111 \\ + 0101 \\ \hline 1100 \end{array}$$

This answer is the correct **binary** result,  $12_{10}$ ; but it is not the proper BCD result. Recall that in BCD,  $12_{10}$  is represented as 0001 0010. Notice that you can obtain the proper BCD result by adding  $0110_2$  to the binary result. The addition of  $0110_2$  is necessary anytime that the binary result exceeds  $1001_2$ .

To produce the proper BCD result when adding two BCD digits, the MPU must follow this procedure:

1. If the sum is  $1001_2$  or less, use the sum as the single digit BCD result.
2. If the sum is greater than  $1001_2$ , add  $0110_2$  and use the result as a 2-digit BCD number.

The situation becomes more complex when packed BCD numbers are added. Consider adding  $0111\ 1001_{BCD}$  ( $79_{10}$ ) to  $0111\ 0011_{BCD}$  ( $73_{10}$ ). The ALU adds these packed BCD numbers as if they were unsigned binary numbers. The result is

$$\begin{array}{r} 11 \quad 1 \\ 0111 \quad 1001 \\ 0111 \quad 0011 \\ \hline 1110 \quad 1100 \end{array}$$

Notice that the result is not a BCD number, since both 4-bit groups exceed  $1001_2$ . Even so, the sum can be converted to BCD by adding  $0110_2$  to each 4-bit group. The result is

$$\begin{array}{r} 1 \quad 1 \quad 1 \quad 1 \\ \quad 1110 \quad 1100 \\ \quad 0110 \quad 0110 \\ \hline 1 \quad 0101 \quad 0010 \end{array}$$

There is a carry from bit 7 that sets the carry bit. This carry bit becomes the most significant BCD digit. Thus, the final BCD result is 0001 0101 0010<sub>BCD</sub> or 152<sub>10</sub>.

If you consider all possible combinations of BCD numbers, you will find that four different situations exist:

1. When some BCD numbers are added, the binary result produced by the ALU is equal to the proper BCD representation. This occurs when both BCD digits of the result are  $1001_2$  or less.
2. The binary sum is adjusted by adding  $06_{16}$  if the least significant BCD digit exceeds  $1001_2$  but the most significant BCD digit does not.
3. The binary sum is adjusted by adding  $60_{16}$  if the most significant BCD digit exceeds  $1001_2$  but the least significant BCD digit does not.
4. The binary sum is adjusted by adding  $66_{16}$  if both BCD digits exceed  $1001_2$ .

While this procedure could be programmed, it would be much better if the MPU performed these operations automatically. Fortunately, our hypothetical microprocessor does this. The programmer simply informs

the MPU that the numbers being added are BCD numbers. The MPU automatically computes the proper BCD result. The way the programmer informs the MPU is via the DAA instruction. When the DAA instruction is placed immediately after an ADD or ADC instruction, the MPU automatically converts the sum to the proper BCD number.

Suppose, for example, that you wish to add two BCD numbers. Assume the numbers are  $3792_{10}$  and  $5482_{10}$ . Naturally, the sum should be  $9274_{10}$ . A program for solving this problem is shown in Figure 4-28. The BCD addend ( $3792_{10}$ ) is in addresses  $0F_{16}$  and  $10_{16}$ . The augend ( $5482_{10}$ ) is in locations  $11_{16}$  and  $12_{16}$ . The BCD sum will be placed in locations  $13_{16}$  and  $14_{16}$ .

HEX ADDRESS	HEX CONTENTS	MNEMONICS/HEX CONTENTS	COMMENTS
00	96	LDA	Load into the accumulator direct
01	10	10	the least significant half of the addend.
02	9B	ADD	Add
03	12	12	the least significant half of the augend.
04	19	DAA	Decimal adjust the sum to BCD.
05	97	STA	Store the result as the
06	14	14	least significant half of the sum.
07	96	LDA	Load
08	0F	0F	the most significant half of the addend.
09	99	ADC	Add
0A	11	11	the most significant half of the augend.
0B	19	DAA	Decimal adjust the sum to BCD.
0C	97	STA	Store the result as the
0D	13	13	most significant half of the sum.
0E	3E	HLT	Halt
0F	37	37	} BCD Addend
10	92	92	
11	54	54	} BCD Augend.
12	82	82	
13	—	—	} Reserved for BCD sum.
14	—	—	

Figure 4-28  
Program for adding  
multiple-precision BCD numbers.

The first two instructions add the least significant halves of the addend and augend. The ADD instruction is followed immediately by the DAA instruction. Therefore, the sum is adjusted to a packed BCD number. The result is stored in location  $14_{16}$  as the lower half of the BCD sum.

Next, the upper halves of the addend and augend are added. This time, the ADC instruction is used because the carry from the previous addition must be added in. Again, the DAA instruction adjusts the sum to BCD. The result is stored as the upper half of the BCD sum.

The DAA instruction must be used properly. It can be used only with addition. Also, it must be used immediately after the addition instruction. It can not be used to convert just any binary number to BCD. It only converts the sum of BCD numbers to the BCD format.

## Self-Test Review

30. How is the ADC instruction different from the ADD instruction?
31. How is the SBC instruction different from the SUB instruction?
32. A primary use of the ADC and SBC instructions is in \_\_\_\_\_ arithmetic.
33. The accumulator contains the number  $7_{10}$ . If two ASLA instructions are executed, what number will be in the accumulator?
34. What is the difference between packed and unpacked BCD numbers?
35. When adding unpacked BCD numbers, under what condition must  $0110_2$  be added to the sum in order to form a BCD sum?
36. What instruction is used to automatically adjust the sum to the proper BCD format when two BCD numbers are added?
37. Can the DAA instruction be used after a SUB instruction to produce the proper BCD difference when two BCD numbers are subtracted?

## Answers

30. When the ADC instruction is executed, an additional 1 is added to the sum if the carry flag is set.
31. When the SBC instruction is executed, an additional 1 is subtracted from the difference if the carry flag is set.
32. Multiple-precision.
33. The first ASLA instruction multiplies the number by two, giving  $14_{10}$ . The second ASLA doubles this number, giving  $28_{10}$ .
34. With packed BCD numbers, each byte contains two BCD digits. With unpacked BCD numbers, each byte contains one BCD digit.
35. When two BCD digits are added,  $0110_2$  must be added to the sum if the sum exceeds  $1001_2$ .
36. The decimal adjust accumulator (DAA) instruction.
37. No. The DAA instruction is used in conjunction with add instructions only.

## EXPERIMENTS

Perform Programming Experiments 5 and 6. You will find these experiments in Unit 9. After you finish these experiments, return to this unit and complete the Unit Examination.

## UNIT EXAMINATION

1. The BRA instruction will cause a branch to occur:
  - A. Anytime that it is executed.
  - B. Only if the Z flag is set.
  - C. Only if the N flag is set.
  - D. Only if the C flag is set.
  
2. The address that follows the opcode of an unconditional branch instruction is:
  - A. The address of the operand.
  - B. The address of the next opcode to be executed.
  - C. Added to the program count to form the address of the next opcode to be executed.
  - D. Added to the program count to form the address of the operand that is to be tested to see if a branch operation is required.
  
3. The opcode for an unconditional branch instruction is at address  $AF_{16}$ . The relative address is  $0F_{16}$ . From what address will the next opcode be fetched?
  - A.  $A0_{16}$ .
  - B.  $C0_{16}$ .
  - C.  $BE_{16}$ .
  - D.  $B1_{16}$ .
  
4. The opcode for an unconditional branch instruction is at address  $30_{16}$ . The relative address is  $EF_{16}$ . From what address will the next opcode be fetched?
  - A.  $21_{16}$ .
  - B.  $EF_{16}$ .
  - C.  $32_{16}$ .
  - D.  $19_{16}$ .
  
5. The carry register:
  - A. Acts like the ninth bit of the accumulator.
  - B. Is set when a "borrow" for bit 7 of the accumulator occurs.
  - C. Is set when a carry from bit 7 occurs.
  - D. All the above.

6. The numbers  $0101\ 1000_2$  and  $0110\ 0011_2$  are added using the ADD instruction. Immediately after the ADD instruction is executed, the condition code registers will indicate the following:
- A.  $C=1, N=1, V=1, Z=0$ .
  - B.  $C=0, N=1, V=1, Z=0$ .
  - C.  $C=0, N=1, V=0, Z=0$ .
  - D.  $C=0, N=0, V=1, Z=1$ .
7. The divide program shown in Figure 4-16 works only if the dividend is initially less than  $+128_{10}$ . The program can be modified to work for dividends up to  $255_{10}$  by replacing the BMI instruction with the:
- A. BEQ instruction.
  - B. BNE instruction.
  - C. BCC instruction.
  - D. BCS instruction.
8. A binary number can be converted to BCD by repeatedly:
- A. Dividing by powers of two.
  - B. Subtracting powers of ten.
  - C. Multiplying by powers of two.
  - D. Adding powers of ten.
9. The DAA instruction is used:
- A. To convert a binary number to BCD.
  - B. To convert a BCD number to binary.
  - C. After an add instruction to adjust the sum to a BCD number.
  - D. After a subtract instruction to adjust the difference to a BCD number.
10. When you are adding multiple-precision binary numbers, all bytes except the least significant ones must be:
- A. Added using the ADD instruction.
  - B. Added using the DAA instruction.
  - C. Added using the ADC instruction.
  - D. Decimal adjusted before addition takes place.





# Individual Learning Program

## MICROPROCESSORS

### *Unit 5*

### THE 6800 MICROPROCESSOR — PART 1

EE-3401

HEATH COMPANY  
BENTON HARBOR, MICHIGAN 49022

Copyright © 1977  
Heath Company  
All Rights Reserved  
Printed in the United States of America

## CONTENTS

Introduction .....	5-3
Unit Objectives .....	5-4
Unit Activity Guide .....	5-5
Architecture of the 6800 MPU .....	5-6
Instruction Set of the 6800 MPU .....	5-15
New Addressing Modes .....	5-36
Experiments .....	5-52
Unit Examination .....	5-53
Examination Answers .....	5-61

## INTRODUCTION

Until now, we have confined our study to a simple hypothetical microprocessor. Obviously, though, this hypothetical model must be very close to the real thing, since we have been running its programs on the ET-3400 Microprocessor Trainer. In this unit, you will begin your study of the actual microprocessors upon which our hypothetical model is based. The microprocessor in the ET-3400 Trainer is called the 6800. It was first released by Motorola in the mid 1970's. Today, it is also supplied by several other companies.

The microprocessor in the ET-3400A Trainer is called the 6808. The primary difference between the 6808 and 6800 microprocessors is the means by which clock signals are generated. The 6808 has an on-chip clock circuit, the 6800 does not. Because of this one difference, there will also be some chip pin assignments that are not identical. These differences will be discussed in more detail in Unit Seven of this course. However, in all other characteristics, such as the instruction set and internal registers, the 6800 and 6808 are identical. It is these identical characteristics which are the subject of this and the following unit. Therefore, this unit refers to the 6800 microprocessor only. But, the data presented also applies to the 6808 in the ET-3400A Trainer.

There are other microprocessors that have similar instruction sets, architecture, and addressing modes. Thus, by becoming familiar with the 6800 (6808), you should be able to understand and use a wide range of microprocessors.

You already know a great deal about the 6800 and/or 6808 microprocessor. You have been programming this device for the past several units. The main difference between the 6800 (6808) microprocessors and our hypothetical model is complexity. As you will see, the 6800 (6808) is a vastly expanded version of our hypothetical model.

## UNIT OBJECTIVES

When you have completed this unit you will be able to:

1. Draw a programming model of the 6800 MPU.
2. Explain the purpose of each block in a simplified block diagram of the 6800 MPU.
3. Using Appendix A and Figure 5-24 as references, explain the operation of all the instructions discussed in this unit.
4. Write simple programs that use indexed and extended addressing.
5. Using Figure 5-24 as a guide, find the opcode, number of MPU cycles, number of bytes, and effects on the condition code flags of every instruction discussed in this unit.

## UNIT ACTIVITY GUIDE

**Completion  
Time**

- |   |       |
|---|-------|
| <input type="checkbox"/> Read Section on Architecture of the 6800 MPU.    | _____ |
| <input type="checkbox"/> Complete Self-Test Review Questions 1 — 8.       | _____ |
| <input type="checkbox"/> Read Section on Instruction Set of the 6800 MPU. | _____ |
| <input type="checkbox"/> Complete Self-Test Review Questions 9 — 26.      | _____ |
| <input type="checkbox"/> Review Appendix A.                               | _____ |
| <input type="checkbox"/> Read Section on New Addressing Modes.            | _____ |
| <input type="checkbox"/> Complete Self-Test Review Questions 27 — 43.     | _____ |
| <input type="checkbox"/> Perform Programming Experiments 7 and 8.         | _____ |
| <input type="checkbox"/> Complete Unit Examination.                       | _____ |
| <input type="checkbox"/> Check Examination Answers.                       | _____ |

## ARCHITECTURE OF THE 6800 MPU

In computer jargon, the word architecture is used to describe the computer's style of construction, its register size and arrangement, its bus configuration, etc. The architecture of our hypothetical microprocessor is shown for one last time in Figure 5-1. By now you should be quite familiar and comfortable with this architecture.

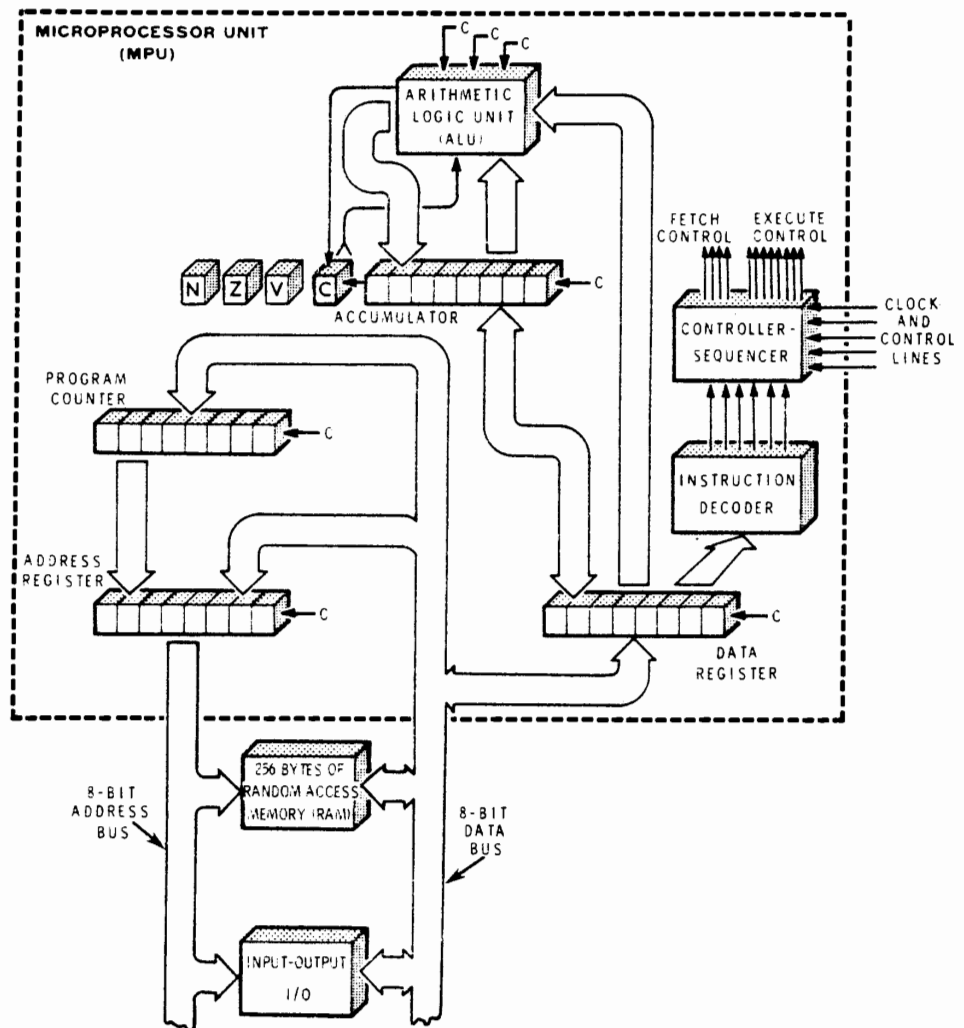


Figure 5-1

Architecture of the hypothetical  
microcomputer.

The only reason for showing the details of the model is to give you an idea of what goes on inside the integrated circuit. In an actual microprocessor the internal structure is often so complex that we become bogged down in details if we attempt to analyze it too closely. For this reason, a programming model is generally used when a microprocessor is being introduced for the first time. In the programming model, the emphasis is shifted upward by an order of magnitude. Any register or circuit that cannot be directly controlled by the programmer is simply ignored. Consider the data register for example. There are no instructions that give the programmer direct control over this register. That is, there are no instructions such as Load Data Register, Store Data Register, etc. All data register activity is controlled strictly by the MPU. Thus, the programmer can simply ignore the existence of this register. The same is true of the address register, the instruction decoder, the controller-sequencer, etc. Therefore, the programming model of our hypothetical MPU can be represented as shown in Figure 5-2. This simple diagram is sufficient for most programming applications since it shows all the registers that can be directly controlled by the program.

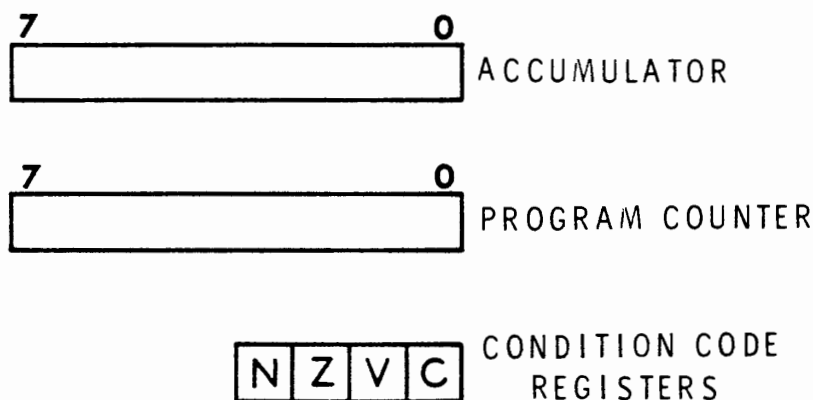
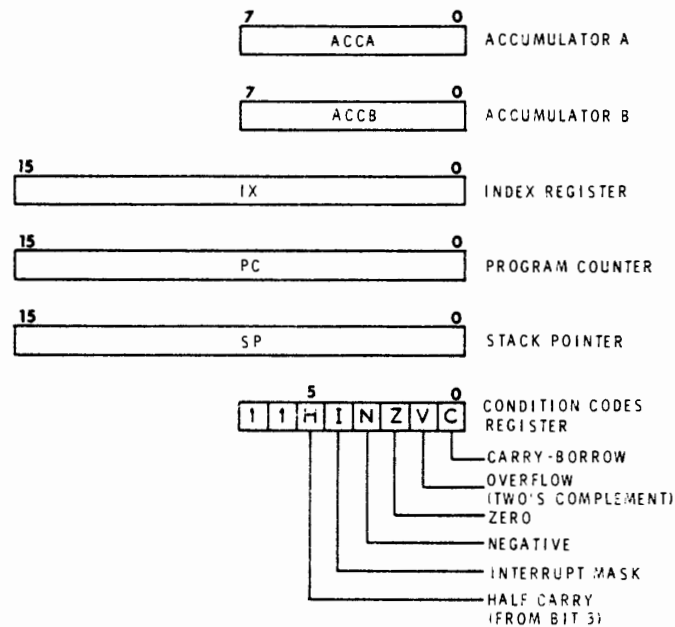


Figure 5-2  
Programming model of the  
hypothetical MPU.

## Programming Model of the 6800 MPU

The 6800 MPU is much more complex than our hypothetical MPU. Consequently, a programming model of the 6800 makes a good starting point. The programming model is shown in Figure 5-3.

**Figure 5-3**  
 Programming model of the  
 6800 MPU.



You will notice immediately that the 6800 MPU has several additional registers. However, only two of these, the index register and the stack pointer, are actually new to you. Let's look at the major differences between this MPU and our hypothetical model.

**Two Accumulators** The 6800 MPU has two accumulators instead of one. They are called accumulator A (ACCA) and accumulator B (ACCB). Each has its own group of instructions associated with it. The names and mnemonics of the instructions specify which accumulator is to be used. Thus, there are instructions such as:

Load Accumulator A (LDAA)  
 Load Accumulator B (LDAB)  
 Store Accumulator A (STAA)  
 Store Accumulator B (STAB)

Notice that a letter is added to both the name and the mnemonic to indicate which accumulator is being used.



From your previous programming experience, you can visualize the value of a second accumulator. For example consider a program in which the MPU counts the number of times that some operation occurs. In the past, we stored the number that the accumulator was presently working on, loaded the count into the accumulator; incremented the count; stored the count; and reloaded the original number. With a second accumulator, none of this is necessary. We can simply maintain the count in accumulator B while working with the number in accumulator A. In fact, we can perform any arithmetic or logic operation on two different numbers without having to shift the numbers back and forth between memory.

**16-Bit Program Counter** The program counter in the 6800 has  $16_{10}$  bits rather than 8. Thus, it can specify  $65,536_{10}$  different addresses. This means that a 6800 based microcomputer can have up to  $65,536_{10}$  bytes of memory. Most applications require substantially less memory than this maximum number. Fortunately, we can use as little or as much memory as we need up to the  $2^{16}$  byte limit.

Since the program counter has  $16_{10}$  bits, the address bus must also be 16-bits wide. Contrast this with the 8-bit address bus of our hypothetical machine.

You may wonder how we specify a 16-bit address with an 8-bit byte. The obvious answer is that two 8-bit bytes are required. Recall that in the direct addressing mode, the address was specified by a single 8-bit byte. The 6800 microprocessor retains this addressing mode. However, since an 8-bit address can specify only  $256_{10}$  addresses, the 6800 MPU can use this mode only if the operand is in the first  $256_{10}$  bytes of memory. To reach higher addresses, a new addressing mode called **extended addressing** must be used. In the extended addressing mode, two bytes are used to represent each address. This addressing mode will be discussed in more detail later. For now, keep in mind that there are  $65,536_{10}$  possible addresses. The lowest address is  $0000_{16}$  and the highest is  $FFFF_{16}$ . Using extended addressing, we have access to any location in memory, but a 2-byte address is required.

**Condition Code Registers** The 6800 MPU has six condition codes. Four of these are almost identical to those discussed in an earlier unit. These include the negative (N), zero (Z), overflow (V) and carry (C) condition codes. The difference arises because there are two accumulators in the 6800 MPU. Thus, the carry flag is set whenever there is a carry from either accumulator. By the same token, an overflow in either accumulator will set the V flag. Later in this unit, you will see how the condition codes are affected by each instruction.

Two new condition codes are shown in Figure 5-3. The I flag is called an interrupt mask. We will discuss this flag later when you study interrupts. The other is called the half carry flag (H). The H flag is set when there is a carry from bit 3 of the accumulator. The MPU uses this flag to determine how to implement the decimal adjust instruction.

These six flags make up bits 0 through 5 of an 8-bit register. Bits 6 and 7 of the condition code register are not used and are always set to 1. Additional details of the condition codes will be brought out as the need arises.

**Index Register** The index register is a special-purpose, 16-bit register that greatly increases the power of the microprocessor. It allows a powerful address mode called indexed addressing. We will examine this addressing mode later in this unit. For now, consider the index register to be just another working register. The fact that it holds two bytes instead of one can be put to good use. The MPU has instructions that allow the index register to be loaded from two adjacent memory bytes. Another instruction allows us to store the contents of the index register in two adjacent memory locations. This allows us to move data in 2-byte groups. Also, the index register can be incremented and decremented. This lets us maintain 16-bit tallies.

**Stack Pointer** The stack pointer is another special-purpose 16-bit register. It allows the MPU and the programmer to use a section of RAM as a last in, first out (LIFO) memory. This capability is extremely valuable when using subroutines or when processing interrupts. These aspects of the stack pointer will be discussed in the next unit. For the time being let's consider the stack pointer to be another 16-bit working register. It too can be loaded from memory, stored in memory, incremented, and decremented.

## Block Diagram of the 6800 MPU

Now that you have seen the programming model of the 6800 MPU, take a look at the block diagram. A simplified block diagram is shown in Figure 5-4. Several data paths, most control lines, and a temporary storage register have been omitted in favor of the major data paths and registers.

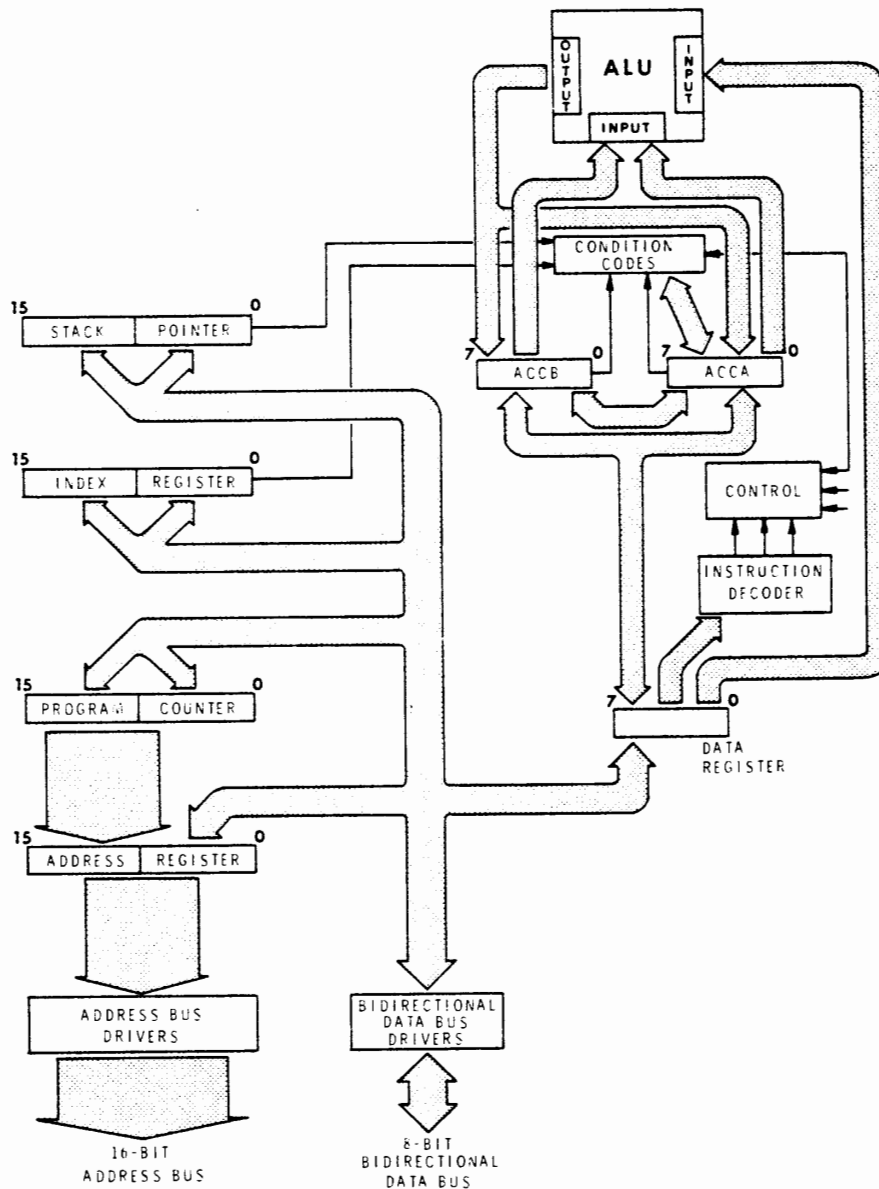


Figure 5-4  
Simplified block diagram of the  
6800 MPU.

The 16-bit registers are shown on the left. These registers are primarily concerned with addressing memory. Since the address bus has 16-bits, all registers associated with addressing must also have 16-bits. Any of the 16-bit registers can be loaded from the data bus. However, because the data bus has only 8-bits, two operations are required to load the 16-bit registers. The upper half of the affected register is always loaded first. Then, a second operation loads the lower half. Although this requires separate MPU cycles, the microprocessor takes care of these operations automatically. For example, a single instruction can load the 16-bit index register with two memory bytes.

The program counter and address register perform exactly the same functions in the 6800 MPU as they did in our hypothetical model. The fetch and execute phases for the immediate and direct addressing modes are virtually identical. The same is true of the relative addressing mode except that the 8-bit relative address is added to the 16-bit program count.

The 8-bit registers are shown on the right. Notice that these circuits are identical to those in our hypothetical model except that there are two accumulators. The condition code registers monitor both accumulators. Also, the two accumulators share the ALU. This allows you to keep track of two separate mathematical operations at more or less the same time. This arrangement is particularly flexible since the contents of one accumulator can be transferred to the other or the contents of the two accumulators can be added together.

## Self-Test Review

1. The microprocessor on which our hypothetical model and the ET-3400 are based is the \_\_\_\_\_ MPU.
2. A major difference between our hypothetical model and the 6800 MPU is that the latter has two \_\_\_\_\_.
3. The program counter in the 6800 MPU has \_\_\_\_\_ bits.
4. How wide is the address bus in a 6800-based microcomputer?
5. What is the range of addresses in the 6800 MPU?
6. List the six condition code flags.
7. Besides the program counter, what other 16-bit registers are used in the 6800 MPU?
8. In the 6800 MPU, does each accumulator have its own carry flag?

## Answers

1. 6800.
2. Accumulators.
3.  $16_{10}$ .
4.  $16_{10}$  bits.
5. From 0000 to  $65,535_{10}$  or 0000 to  $FFFF_{16}$ .
6. Carry — borrow (C)  
Overflow (V)  
Zero (Z)  
Negative (N)  
Interrupt Mask (I)  
Half Carry (H)
7. Index register and stack pointer.
8. No, the two accumulators share a common carry flag.

## INSTRUCTION SET OF THE 6800 MPU

The 6800 MPU has about  $100_{10}$  basic instructions. Moreover, when all the different addressing modes are considered, there are  $197_{10}$  different opcodes to which the MPU will respond.

These instructions can be broken down into seven general categories. While some of the categories overlap, the general classifications of instructions are: arithmetic, data handling, logic, data test, index register and stack pointer, jump and branch, and condition code. In this unit we will discuss most of these instructions in detail. The handful of instructions that are not discussed in this unit will be described in the following unit.

In this section we will not be concerned with addressing modes. Therefore, no opcodes are given. Later, we will look at the various addressing modes and opcodes. For now, though, let's identify the instructions by their names, mnemonics, and operations. You will see what each instruction does and how it affects the various condition code registers.

Because of the large number of instructions covered in this section, the explanations will be general and brief. You are **not** expected to remember all the details of every instruction. Appendix A of this course contains a detailed listing of each instruction. It explains every detail of the various instructions. After reading this section, turn to Appendix A and look over the explanations given there. In the future, when you are in doubt as to exactly what a particular instruction does, look it up in Appendix A.

## Arithmetic Instructions

Figure 5-5 shows the arithmetic instructions of the 6800 MPU. The name of each instruction is given on the left. The next column contains the mnemonics. The center column gives a shorthand description of what the instruction does. The right-hand columns show how the various condition code registers are affected.

ACCUMULATOR AND MEMORY		BOOLEAN/ARITHMETIC OPERATION (All register labels refer to contents)	COND. CODE REG.					
			5	4	3	2	1	0
			H	I	N	Z	V	C
Add	ADDA	$A + M \rightarrow A$	↑	●	↑	↑	↑	↑
	ADDB	$B + M \rightarrow B$	↑	●	↑	↑	↑	↑
Add Acmltrs	ABA	$A + B \rightarrow A$	↑	●	↑	↑	↑	↑
Add with Carry	ADCA	$A + M + C \rightarrow A$	↑	●	↑	↑	↑	↑
	ADCB	$B + M + C \rightarrow B$	↑	●	↑	↑	↑	↑
Complement, 2's (Negate)	NEG	$00 - M \rightarrow M$	●	●	↑	↑	①	②
	NEGA	$00 - A \rightarrow A$	●	●	↑	↑	①	②
	NEGB	$00 - B \rightarrow B$	●	●	↑	↑	①	②
Decimal Adjust, A	DAA	Converts Binary Add. of BCD Characters into BCD Format*	●	●	↑	↑	↑	③
Subtract	SUBA	$A - M \rightarrow A$	●	●	↑	↑	↑	↑
	SUBB	$B - M \rightarrow B$	●	●	↑	↑	↑	↑
Subtract Acmltrs.	SBA	$A - B \rightarrow A$	●	●	↑	↑	↑	↑
Subtr. with Carry	SBCA	$A - M - C \rightarrow A$	●	●	↑	↑	↑	↑
	SBCB	$B - M - C \rightarrow B$	●	●	↑	↑	↑	↑

\*Used after ABA, ADC, and ADD in BCD arithmetic operation; each 8-bit byte regarded as containing two 4-bit BCD numbers. DAA adds 0110 to lower half-byte if least significant number >1001 or if preceding instruction caused a Half-carry. Adds 0110 to upper half-byte if most significant number >1001 or if preceding instruction caused a Carry. Also adds 0110 to upper half-byte if least significant number >1001 and most significant number = 9.

(Bit set if test is true and cleared otherwise)

- ① (Bit V) Test: Result = 10000000?
- ② (Bit C) Test: Result = 00000000?
- ③ (Bit C) Test: Decimal value of most significant BCD Character greater than nine?  
(Not cleared if previously set.)

Figure 5-5  
Arithmetic instructions.



To be certain you have the idea, let's go through the first instruction in detail. The first instruction is the add instruction. Actually, since the 6800 has two accumulators, there are two add instructions. Their mnemonics are ADDA and ADDB. Notice that the final letter of the mnemonic indicates which accumulator (A or B) is involved. The shorthand representation of the operation is:  $A + M \rightarrow A$ . The note at the top of this column tells you that the register labels refer to the contents of the register. Thus, A means the contents of accumulator A and M means the contents of the affected memory location. The symbol ( $\rightarrow$ ) means "Transfer into." Therefore,  $A + M \rightarrow A$  means "Add the contents of accumulator A to the contents of the affected memory location and transfer the sum into accumulator A."

To see how the condition code flags are affected, you simply look over to the right under whatever condition code you are interested in. Generally, the condition code is either unaffected or is tested and set accordingly. When the condition code is unaffected, this is represented by the symbol ( $\bullet$ ). For example, none of the arithmetic instructions affect the I flag. Most of the arithmetic instructions test the condition codes and set them if the condition exists. For example, if the result of an arithmetic operation is zero, the Z flag is set to 1. If this condition does not exist, the Z flag is reset or cleared to 0. The symbol ( $\updownarrow$ ) means "test and set if true; clear otherwise." Occasionally, a note is necessary to describe some unusual situation regarding the condition code. This is represented by a number within a circle. The notes are given at the bottom of the drawing.

The ADDA and ADDB instructions are self-explanatory. The ABA instruction adds the contents of accumulator A to the contents of accumulator B. The result is stored in accumulator A.

The add with carry instructions are identical to those discussed earlier for our hypothetical machine. Notice that the carry bit is added in with the sum.

Because two's complement arithmetic is used in the 6800 MPU, instructions are provided that allow us to take the two's complement of a number. The negate instruction subtracts the contents of the affected register from  $00_{16}$ . This is the same as taking the two's complement of the number. The affected register can be any memory location (M) or either accumulator (A or B). Thus, there are three different negate instructions. Keep in mind that NEG means "take the two's complement of the affected memory location;" NEGA means "take the two's complement of accumulator A;" etc.

Notice that the NEG instruction allows us to operate on a byte in memory without first fetching the operand from memory. In the past, we have loaded the operand, performed the operation, and then stored the new operand. However, the 6800 allows us to perform certain operations on the operand without first fetching it from memory. Several examples of this will be pointed out as we progress through the instruction set.

The decimal adjust instruction performs exactly as it did in our hypothetical machine. The note immediately under the table summarizes its operation. It must also be pointed out that this instruction works only with accumulator A.

The subtract and the subtract with carry instructions are self-explanatory. They perform as described earlier for our hypothetical MPU. The 6800 MPU has an additional subtract instruction. The SBA instruction subtracts the contents of accumulator B from the contents of accumulator A. The resulting difference is placed in accumulator A.

## Data Handling Instructions

Figure 5-6 shows the largest group of instructions used by the 6800 MPU. These can be loosely categorized as data handling instructions.

The clear instructions allow us to clear a memory location or either accumulator. In the past, we have cleared bytes of memory by first clearing the accumulator and then storing the resulting  $00_{16}$  in the proper memory location. However, the CLR instruction allows us to clear a memory location with a single instruction. Notice that some new entries appear in the condition code registers column. R means that the condition code is always reset or cleared to 0. S means that the code is always set to 1.

The decrement instruction allows us to subtract 1 from a memory location or from either accumulator. The DEC instruction is especially valuable since it allows us to decrement a byte in memory with a single instruction. Previously we have loaded the byte, decremented it, and then stored it back in memory.

The increment instructions are similar except they allow us to add 1 to a memory location or one of the accumulators. Notice that the INC instruction allows us to maintain a tally in memory without having to load it, increment it, and then store it away.

The load accumulator instructions are self-explanatory. Notice that either accumulator can be loaded from memory.

ACCUMULATOR AND MEMORY		BOOLEAN/ARITHMETIC OPERATION (All register labels refer to contents)	COND. CODE REG.					
			5	4	3	2	1	0
			H	I	N	Z	V	C
Clear	CLR	00 → M	•	•	R	S	R	R
	CLRA	00 → A	•	•	R	S	R	R
	CLRB	00 → B	•	•	R	S	R	R
Decrement	DEC	M - 1 → M	•	•	↑	↑	④	•
	DECA	A - 1 → A	•	•	↑	↑	④	•
	DECB	B - 1 → B	•	•	↑	↑	④	•
Increment	INC	M + 1 → M	•	•	↑	↑	⑤	•
	INCA	A + 1 → A	•	•	↑	↑	⑤	•
	INCB	B + 1 → B	•	•	↑	↑	⑤	•
Load Acmltr	LDAA	M → A	•	•	↑	↑	R	•
	LDAB	M → B	•	•	↑	↑	R	•
Rotate Left	ROL	M	•	•	↑	↑	⑥	↑
	ROLA	A	•	•	↑	↑	⑥	↑
	ROLB	B	•	•	↑	↑	⑥	↑
Rotate Right	ROR	M	•	•	↑	↑	⑥	↑
	RORA	A	•	•	↑	↑	⑥	↑
	RORB	B	•	•	↑	↑	⑥	↑
Shift Left, Arithmetic	ASL	M	•	•	↑	↑	⑥	↑
	ASLA	A	•	•	↑	↑	⑥	↑
	ASLB	B	•	•	↑	↑	⑥	↑
Shift Right, Arithmetic	ASR	M	•	•	↑	↑	⑥	↑
	ASRA	A	•	•	↑	↑	⑥	↑
	ASRB	B	•	•	↑	↑	⑥	↑
Shift Right, Logic	LSR	M	•	•	R	↑	⑥	↑
	LSRA	A	•	•	R	↑	⑥	↑
	LSRB	B	•	•	R	↑	⑥	↑
Store Acmltr	STAA	A → M	•	•	↑	↑	R	•
	STAB	B → M	•	•	↑	↑	R	•
Transfer Acmltrs	TAB	A → B	•	•	↑	↑	R	•
	TBA	B → A	•	•	↑	↑	R	•

- ④ (Bit V) Test: Operand = 10000000 prior to execution?  
 ⑤ (Bit V) Test: Operand = 01111111 prior to execution?  
 ⑥ (Bit V) Test: Set equal to result of  $N \oplus C$  after shift has occurred.

Figure 5-6  
Data handling instructions.

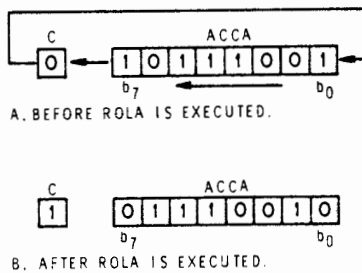


Figure 5-7

Executing the ROLA instruction.

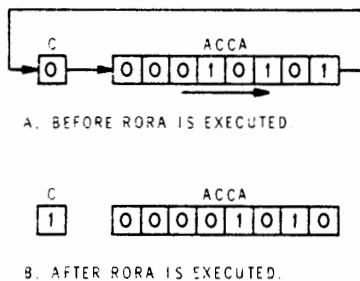


Figure 5-8

Executing the RORA instruction.

The rotate left instructions allow us to shift the contents of the accumulator or a memory location without losing bits of data. Consider the ROLA instruction as an example. When this instruction is executed, the A accumulator and the carry bit form a 9-bit circulating register. That is, they form a closed loop as shown in Figure 5-7A. When ROLA is executed, the data is rotated clockwise. The MSB of A shifts into the carry register. Simultaneously, the contents of A are shifted left. Notice that the carry bit is not lost. Instead it is shifted into the LSB of the accumulator.

While the usefulness of this instruction may not be obvious, it is a valuable tool. For example, it could be used to determine parity. By repeatedly rotating left and testing the C flag, you could determine the number of 1's in the byte. Once you know this, you could easily generate the proper parity bit.

The ROL instruction allows you to rotate a memory byte to the left while it is still in memory. ROLB allows you to rotate the B accumulator to the left. In each case, the C register is used as a ninth bit.

The rotate right instructions are identical except that the direction of rotation is reversed. Figure 5-8 illustrates the execution of the RORA instruction. This instruction is also valuable. Suppose for example that we wish to know if the number in the accumulator is even or odd. This is determined by the LSB of the number. If LSB = 1, the number is odd; if LSB = 0, the number is even. One way to determine this is to rotate the number to the right so that the LSB is in the C register. We could then test the C register to see if it is set or cleared. Notice that the number could then be restored to its original value by the ROLA instruction.

The arithmetic shift left instruction was discussed earlier in our hypothetical MPU. The ASLA instruction performs exactly as described in the previous unit. However, notice that the 6800 MPU also has an ASLB instruction that performs the same operation with accumulator B. Also, it has an ASL instruction that allows us to perform this operation on a byte that is in memory. Figure 5-9 illustrates the execution of this instruction.

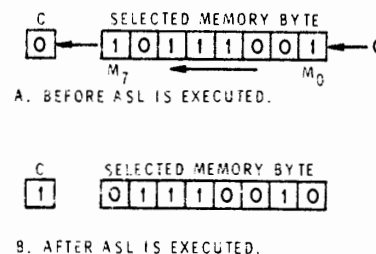


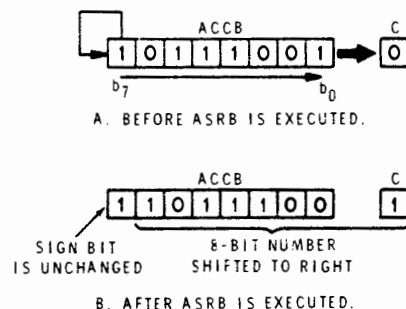
Figure 5-9

Executing the ASL instruction.

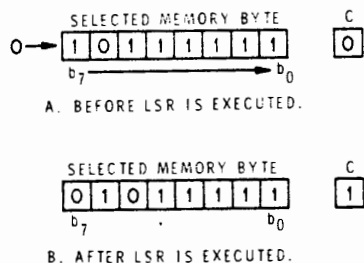
While there is only one type of shift left instruction, there are two types of shift right instructions. Let's discuss the arithmetic shift right instructions first.

When an **arithmetic** shift right instruction is executed, the number in the affected register is shifted right one position. The LSB goes into the C register.  $B_1$  shifts to  $B_0$ , etc.  $B_7$  shifts into  $B_6$ . However,  $B_7$  itself remains unchanged. Figure 5-10 illustrates the execution of the ASRB instruction. Notice that there are also ASRA and ASR instructions listed in Figure 5-6. These perform the same type of shift operation but on accumulator A and the selected memory byte respectively.

The logic shift right instructions are different in that they do not retain the sign bit. When a logic shift right is executed, the contents of the affected register are shifted to the right. The LSB goes into the carry register. The MSB is filled with a 0. For example, the LSR instruction is illustrated in Figure 5-11. While this instruction shifts the selected memory locations, LSRA and LSRB can be used to perform similar operations on accumulators A and B respectively.



**Figure 5-10**  
Executing the ASRB instruction.



**Figure 5-11**  
Executing the LSR instruction.

Referring back to Figure 5-6, the store accumulator instructions are self-explanatory.

The final data handling instructions are the transfer accumulator instructions. TAB copies the contents of accumulator A into accumulator B. After this instruction is executed, the number originally in accumulator A will be in both accumulators. TBA does just the opposite. It copies the contents of accumulator B into accumulator A. After TBA is executed, the number originally in accumulator B will be in both accumulators.

## Logic Instructions

The logic instructions in the 6800 MPU are similar to those in our hypothetical MPU. Figure 5-12 shows the 6800's logic instructions.

ACCUMULATOR AND MEMORY		BOOLEAN/ARITHMETIC OPERATION (All register labels refer to contents)	COND. CODE REG.					
			5	4	3	2	1	0
			H	I	N	Z	V	C
And	ANDA	$A \bullet M \rightarrow A$	•	•	↑	↑	R	•
	ANDB	$B \bullet M \rightarrow B$	•	•	↑	↑	R	•
Complement, 1's	COM	$\bar{M} \rightarrow M$	•	•	↑	↑	R	S
	COMA	$\bar{A} \rightarrow A$	•	•	↑	↑	R	S
	COMB	$\bar{B} \rightarrow B$	•	•	↑	↑	R	S
Exclusive OR	EORA	$A \oplus M \rightarrow A$	•	•	↑	↑	R	•
	EORB	$B \oplus M \rightarrow B$	•	•	↑	↑	R	•
Or, Inclusive	ORA	$A \uparrow M \rightarrow A$	•	•	↑	↑	R	•
	ORB	$B \uparrow M \rightarrow B$	•	•	↑	↑	R	•

Figure 5-12

Logic instructions.

There is one AND instruction for each accumulator. The contents of the specified accumulator are ANDed bit-for-bit with the contents of the selected memory location. The result is placed back in the accumulator. This is identical to the AND instruction in our hypothetical machine.

The complement instructions allow you to take the 1's complement of the number in the affected register. COM allows you to complement a byte in memory.

COMA and COMB allow you to complement the contents of accumulators A and B respectively. In each case, all 1's are changed to 0's and all 0's are changed to 1's.

The exclusive OR instructions work like the one in our hypothetical MPU. The contents of the specified accumulator are exclusively ORed bit-for-bit with the contents of the selected memory location. The result is stored back in the specified accumulator.

The inclusive OR is similar except that the contents of the specified accumulator are inclusively ORed with the contents of the selected memory location.

## Data Test Instructions

These are a powerful group of instructions that allow us to compare operands in several different ways. In previous units, you had experience comparing operands. The most frequently used method was to subtract one operand from another and test the result for zero or negative. In many cases, the numeric result of the subtraction was unimportant. We needed to know only if the result was zero or minus. The data test instructions allow us to make several different tests without actually producing an unwanted numeric result. These instructions are shown in Figure 5-13.

ACCUMULATOR AND MEMORY		BOOLEAN/ARITHMETIC OPERATION (All register labels refer to contents)	COND. CODE REG.					
			5	4	3	2	1	0
			H	I	N	Z	V	C
Bit Test	BITA	A • M	•	•	↑	↑	R	•
	BITB	B • M	•	•	↑	↑	R	•
Compare	CMPA	A - M	•	•	↑	↑	↑	↑
	CMPB	B - M	•	•	↑	↑	↑	↑
Compare Acmltrs	CBA	A - B	•	•	↑	↑	↑	↑
Test, Zero or Minus	TST	M - 00	•	•	↑	↑	R	R
	TSTA	A - 00	•	•	↑	↑	R	R
	TSTB	B - 00	•	•	↑	↑	R	R

Figure 5-13

Data test instructions.

The bit test instructions are very similar to the AND instructions. In both cases, the contents of the specified accumulator are ANDed with the contents of the selected memory location. The difference is that with the bit test instruction no logical product is produced. Neither the contents of the accumulator nor memory are altered in any way. However, the condition code registers are affected just as if the AND operation had taken place. Consider the BITA instruction. When executed, A is ANDed with M. If the result is 00<sub>16</sub>, the Z register is set. Otherwise, the Z register is cleared. If the MSB of the result is 1, the N flag is set. However, the contents of the accumulator and memory are unaffected.

In the same way, the compare instructions are similar to subtract instructions except that the resulting numeric difference is ignored. For example, when the CMPA instruction is executed, the contents of the selected memory location are subtracted from the contents of accumulator A. The condition codes are affected just as if a difference had been produced. However, the original contents of accumulator A and memory are unaffected.

The compare accumulators instruction (CBA) works the same way. The condition codes are set as if the contents of B were subtracted from the contents A. However, the contents of the accumulators are unaffected.

Finally, the test for zero or minus instruction allows you to test the number in one of the accumulators or the memory to see if it is negative or zero. When this instruction is executed, the MPU looks at the number in question and sets the N and Z flags accordingly. The number itself is not changed.

## Index Register and Stack Pointer Instructions

The index register and stack pointer are 16-bit registers. Figure 5-14 shows eleven instructions that allow us to control the operation of these registers. Because of the 16-bit format, the load, store, and compare instructions are slightly different from those discussed earlier.

INDEX REGISTER AND STACK			5	4	3	2	1	0
POINTER OPERATIONS	MNEMONIC	BOOLEAN/ARITHMETIC OPERATION	H	I	N	Z	V	C
Compare Index Reg	CPX	$(X_H/X_L) - (M/M + 1)$	•	•	①	↑	②	•
Decrement Index Reg	DEX	$X - 1 \rightarrow X$	•	•	•	↑	•	•
Decrement Stack Pntr	DES	$SP - 1 \rightarrow SP$	•	•	•	•	•	•
Increment Index Reg	INX	$X + 1 \rightarrow X$	•	•	•	↑	•	•
Increment Stack Pntr	INS	$SP + 1 \rightarrow SP$	•	•	•	•	•	•
Load Index Reg	LDX	$M \rightarrow X_H, (M + 1) \rightarrow X_L$	•	•	③	↑	R	•
Load Stack Pntr	LDS	$M \rightarrow SP_H, (M + 1) \rightarrow SP_L$	•	•	③	↑	R	•
Store Index Reg	STX	$X_H \rightarrow M, X_L \rightarrow (M + 1)$	•	•	③	↑	R	•
Store Stack Pntr	STS	$SP_H \rightarrow M, SP_L \rightarrow (M + 1)$	•	•	③	↑	R	•
Indx Reg $\rightarrow$ Stack Pntr	TXS	$X - 1 \rightarrow SP$	•	•	•	•	•	•
Stack Pntr $\rightarrow$ Indx Reg	TSX	$SP + 1 \rightarrow X$	•	•	•	•	•	•

- ① (Bit N) Test: Sign bit of most significant (MS) byte of result = 1?  
 ② (Bit V) Test: 2's complement overflow from subtraction of LS bytes?  
 ③ (Bit N) Test: Result less than zero? (Bit 15 = 1)

Figure 5-14  
 Index register and stack pointer  
 instructions.



The compare index register (CPX) instruction allows us to compare the 16-bit number in the index register with any two consecutive bytes in memory. Recall that the index register (X) will hold two bytes. The higher byte is identified as  $X_H$  while the lower byte is called  $X_L$ . When the CPX instruction is executed,  $X_H$  is compared with the 8-bit byte in the specified memory location (M). Also,  $X_L$  is compared with the byte immediately following the specified memory location (M+1). The comparison is the same as if M and M+1 were subtracted from  $X_H$  and  $X_L$  except that no numeric difference is produced. Neither X nor M is changed in any way. However, the N, Z, and V condition codes are affected as shown in Figure 5-14. Generally, the Z code is the one we are interested in since it tells us whether or not an exact match exists between the index register and the two bytes in memory.

The next four instructions are self-explanatory. They allow us to increment and decrement either the index register or the stack pointer. For one thing, these instructions allow us to maintain two separate 16-bit tallies simultaneously. However, the real value of these instructions and their associated registers will be discussed later.

The load and store instructions for the 16-bit registers are shown next in Figure 5-14. Since these are two byte registers, the LDX and LDS instructions must load two bytes from memory. In the case of the index register, the specified memory byte (M) is loaded into the upper half of the index register ( $X_H$ ). An instant later, the next byte in memory (M+1) is automatically loaded into the lower half of the index register ( $X_L$ ). Thus, the operation can be described as:  $M \rightarrow X_H, (M+1) \rightarrow X_L$ .

Because the stack pointer is also a 16-bit register, the load stack pointer instruction (LDS) works the same way. Its operation can be described as:  $M \rightarrow SP_H, (M+1) \rightarrow SP_L$ . Here,  $SP_H$  refers to the upper half of the stack pointer while  $SP_L$  refers to the lower half.

When the contents of the 16-bit registers are being stored, the operation is reversed. For example, the STX instruction stores  $X_H$  in M and  $X_L$  in M+1. A similar instruction, STS, allows us to store the contents of the stack pointer in the same way.

The final two instructions in this group allow us to transfer numbers between these two 16-bit registers. The TXS instruction loads the stack pointer with the contents of the index register **minus one**. The TSX instruction loads the index register with the contents of the stack pointer **plus one**. A more detailed discussion of these two important registers and their associated instructions will be given in the next unit.

## Branch Instructions

The branch instructions are shown in Figure 5-15. Two additional instructions are also thrown in since they affect the program counter.

BRANCH			5	4	3	2	1	0
OPERATIONS	MNEMONIC	BRANCH TEST	H	I	N	Z	V	C
Branch Always	BRA	None	•	•	•	•	•	•
Branch If Carry Clear	BCC	$C = 0$	•	•	•	•	•	•
Branch If Carry Set	BCS	$C = 1$	•	•	•	•	•	•
Branch If = Zero	BEQ	$Z = 1$	•	•	•	•	•	•
Branch If $\geq$ Zero	BGE	$N \oplus V = 0$	•	•	•	•	•	•
Branch If $>$ Zero	BGT	$Z + (N \oplus V) = 0$	•	•	•	•	•	•
Branch If Higher	BHI	$C + Z = 0$	•	•	•	•	•	•
Branch If $\leq$ Zero	BLE	$Z + (N \oplus V) = 1$	•	•	•	•	•	•
Branch If Lower Or Same	BLS	$C + Z = 1$	•	•	•	•	•	•
Branch If $<$ Zero	BLT	$N \oplus V = 1$	•	•	•	•	•	•
Branch If Minus	BMI	$N = 1$	•	•	•	•	•	•
Branch If Not Equal Zero	BNE	$Z = 0$	•	•	•	•	•	•
Branch If Overflow Clear	BVC	$V = 0$	•	•	•	•	•	•
Branch If Overflow Set	BVS	$V = 1$	•	•	•	•	•	•
Branch If Plus	BPL	$N = 0$	•	•	•	•	•	•
No Operation	NOP	Advances Prog. Cntr. Only	•	•	•	•	•	•
Wait for Interrupt	WAI		•	①	•	•	•	•

① (Bit 1) Set when interrupt occurs. If previously set, a Non-Maskable Interrupt is required to exit the wait state.

Figure 5-15  
Jump and branch instructions.

Nine of these instructions were discussed in the previous unit. These are: Branch Always (BRA); Branch If Carry Clear (BCC); Branch If Carry Set (BCS); Branch If Equal Zero (BEQ); Branch If Not Equal Zero (BNE); Branch If Minus (BMI); Branch If Plus (BPL); Branch If Overflow Clear (BVC); and Branch If Overflow Set (BVS).

Before we discuss the new branch instructions, here are some of the symbols we will be using. The symbol ( $\geq$ ) means "is greater than or is equal to"; ( $>$ ) means "is greater than"; ( $\leq$ ) means "is less than or is equal to"; ( $<$ ) means "is less than"; and ( $\neq$ ) means "is not equal to."

Now consider the Branch If Greater Than or Equal instruction (BGE). This instruction is normally used after a subtract or compare instruction. It will cause a branch operation if the two's complement value in the accumulator is greater than or equal to the two's complement operand in memory. This condition is indicated by the N and V flags having the same value. The MPU determines if this condition is met by exclusively ORing N and V and examining the result.

Three simple examples may help illustrate the operation of this instruction. Let's start with a number in the accumulator that is greater than the operand in memory:

Number in Accumulator	=	00000010 <sub>2</sub>
Operand in Memory	=	00000001 <sub>2</sub>

When the operand is subtracted, the result is 00000001<sub>2</sub>. With this result, both N and V are cleared to 0. Notice that N and V are equal and  $N \oplus V = 0$ . If the BGE instruction followed the subtract operation, the branch would be implemented.

Now see what happens when the number in the accumulator is equal to the operand:

Number in Accumulator	=	00000010 <sub>2</sub>
Operand in Memory	=	00000010 <sub>2</sub>

When the operand is subtracted, the result is 00000000<sub>2</sub>. Again N and V are cleared to 0. Thus, N and V are still equal and  $N \oplus V = 0$ . Again, the BGE instruction would cause a branch to occur.

Finally, note what happens when the number in the accumulator is smaller:

Number in Accumulator	=	00000001 <sub>2</sub>
Operand in Memory	=	00000010 <sub>2</sub>

When the operand is subtracted, the result is 11111111<sub>2</sub>. This time N is set but V is cleared. Thus, N and V are not equal. Therefore,  $N \oplus V = 1$ . In this case, the BGE conditions are not met and no branch will occur. The branch occurs if the two's complement value in the accumulator is greater than or equal to the two's complement operand in memory.

Next, consider the Branch If Greater Than (BGT) instruction. This instruction is normally used immediately after a subtract or compare operation. The branch will occur only if the two's complement minuend was greater than the two's complement subtrahend. By trying several examples as we did above, you will find that the branch conditions are met when  $Z = 0$  and  $N = V$ .

The Branch If Higher (BHI) instruction is similar to the BGT instruction except that it is concerned with **unsigned** numbers. BHI is normally used after a subtract or compare operation. The branch will occur only if the unsigned minuend was greater than the unsigned subtrahend. By trying several different examples, you can prove that this occurs only when the C and Z flags are both 0.

The Branch If Less Than or Equal (BLE) instruction allows you to compare two's complement numbers in another way. If it is executed immediately after a subtract or compare operation, the branch will occur only if the two's complement minuend was less than or equal to the two's complement subtrahend.

The Branch If Lower Or Same (BLS) instruction is similar to the BLE instruction except that **unsigned** numbers are compared. When it is executed immediately after a subtract or compare operation, the branch will occur only if the unsigned minuend was lower than or equal to the unsigned subtrahend.

The Branch If Less Than Zero (BLT) instruction is also similar to the BLE instruction except that the equal qualification is removed. If BLT is executed immediately after a subtract or compare operation, the branch occurs only if the two's complement minuend was less than the two's complement subtrahend.

Two additional instructions are included in Figure 5-15. Although they are not branch instructions, they are included here since they do not seem to fit any of the other categories.

The No Operation (NOP) instruction is a "do-nothing" instruction that simply consumes a small increment of time. It does not change the contents of any register except the program counter. It does increment the program counter by one and consumes two MPU cycles. In spite of this, the NOP is a very useful instruction. When writing a program, we frequently use too many instructions. Once the program is loaded in memory, it is often inconvenient to simply remove an instruction. The hole left in memory can be filled by moving back all instructions that follow. However, a faster way is to simply fill the hole with one or more NOP instructions.

The Wait For Interrupt (WAI) instruction is the 6800's version of a HLT instruction. In earlier units we used this instruction at the end of all our programs. We will continue to use it in the same manner in the future. However, as you will see in the next unit, there is more involved in executing the WAI instruction than simply stopping the MPU. For now, though, continue to think of the WAI as a simple halt instruction.

## Condition Code Register Instructions

The 6800 MPU has eight instructions that allow us direct access to the condition codes. These are listed in Figure 5-16.

CONDITION CODE REGISTER			5	4	3	2	1	0
OPERATIONS	MNEMONIC	BOOLEAN OPERATION	H	I	N	Z	V	C
Clear Carry	CLC	$0 \rightarrow C$	•	•	•	•	•	R
Clear Interrupt Mask	CLI	$0 \rightarrow I$	•	R	•	•	•	•
Clear Overflow	CLV	$0 \rightarrow V$	•	•	•	•	R	•
Set Carry	SEC	$1 \rightarrow C$	•	•	•	•	•	S
Set Interrupt Mask	SEI	$1 \rightarrow I$	•	S	•	•	•	•
Set Overflow	SEV	$1 \rightarrow V$	•	•	•	•	S	•
Accmltr A $\rightarrow$ CCR	TAP	$A \rightarrow CCR$	①					
CCR $\rightarrow$ Accmltr A	TPA	$CCR \rightarrow A$	•	•	•	•	•	•

R = Reset

S = Set

• = Not affected

① (ALL) Set according to the contents of Accumulator A.

Figure 5-16

Condition code register instructions.

Most of these instructions are self-explanatory. The Clear Carry (CLC) instruction resets the C flag to 0 while the Set Carry (SEC) sets it to 1. In the same way, the CLV and SEV instructions allow us to clear and set the overflow flag. Also, the CLI and SEI instructions can be used to clear or set the interrupt flag.

You will notice that there are no instructions for individually clearing the N, Z, or H flags. However, we can still set or clear these flags with the Transfer Accumulator A to the Processor Status Register (TAP) instruction. Figure 5-17 illustrates the execution of this instruction. The contents of bits 0 through 5 of accumulator A are transferred to the condition code registers. Thus, this instruction allows us to set or clear all the condition codes at once.

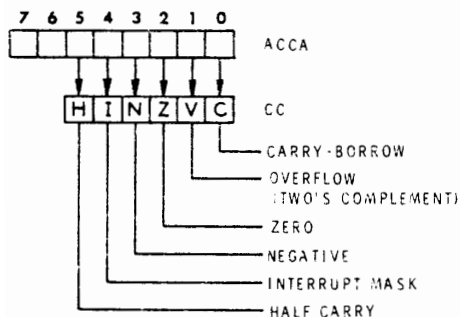


Figure 5-17

Executing the TAP instruction.

The final instruction is the Transfer Processor Status to Accumulator A (TPA) instruction. When this instruction is executed, the contents of the condition code registers are transferred to bits 0 through 5 of accumulator A. This operation is illustrated in Figure 5-18. Notice that bits 6 and 7 of the accumulator are set to 1.

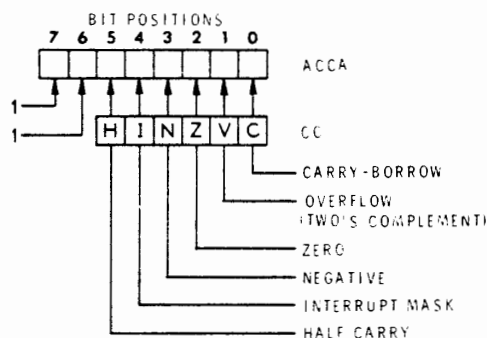


Figure 5-18  
Executing the TPA instruction.

## Summary of Instruction Set

As you can see, the 6800 MPU has a wide variety of instructions. In this section, most of the instructions have been mentioned briefly. However, a full explanation of some instructions must wait until additional new concepts have been covered.

In one short section, it is very difficult to cover every instruction in detail. And, it is virtually impossible for the reader to remember all the details of each instruction. Remember, all of the instructions available to the 6800 MPU are explained in detail in Appendix A of this program. Also, they are arranged alphabetically by their mnemonics for easy reference. Refer to Appendix A any time you are in doubt about what an instruction does. Be sure to look over the introductory material in the Appendix so that you understand all the conventions and symbols.

## Self-Test Review

9. List the seven general categories of instructions.
10. What is meant by the shorthand notation:  $A+B \rightarrow A$ ?
11. How is the C flag affected by the "add" and "add with carry" instructions?
12. Is the C flag changed when the AND instruction is executed?
13. Explain the difference between the NEG instruction and the COM instruction.
14. Explain the difference between the ANDA instruction and the BITA instruction.
15. The decimal adjust instruction is associated with which accumulator?
16. When the RORA instruction is executed the LSB of accumulator A is shifted into the \_\_\_\_\_ register.
17. List eleven operations that can be performed directly to an operand in memory without first loading it into one of the MPU registers.



18. Explain the difference between the SUBB instruction and the CMPB instruction.
19. List the four types of logic operations that the 6800 MPU can perform.
20. When the LDX instruction is executed, from where is the index register loaded?
21. List four conditional branch instructions that are commonly used after a compare or subtract instruction to compare two's complement numbers.
22. Explain the difference between the BGT and BHI instructions.
23. Which instruction is often used to fill in a hole left in a program after an unwanted byte is removed?
24. Which instruction in the 6800 roughly corresponds to the halt instruction in our hypothetical machine?
25. Which of the condition codes can be individually set or cleared?
26. When you have some doubt as to exactly what operation is performed by a given instruction, where can you look to find the answer?

## Answers

9. Arithmetic, data handling, logic, data test, index register and stack pointer, jump and branch, condition code.
10. Add the contents of accumulator A to the contents of accumulator B; transfer the result to accumulator A.
11. The C flag is set if a carry occurs; it is cleared otherwise.
12. No, the C flag is unaffected by the AND instruction.
13. The COM instruction replaces the operand with its 1's complement. The NEG instruction replaces the operand with its 2's complement.
14. With the ANDA instruction, the result of the AND operation is placed in accumulator A. With the BITA instruction, the condition code registers are set according to the result but the result is not retained.
15. The decimal adjust instruction works only with the A accumulator.
16. Carry (C).
17. A byte in memory can be: cleared, incremented, decremented, complemented, negated, rotated left, rotated right, shifted left arithmetically, shifted right arithmetically, shifted right logically, and tested.

18. With the SUBB instruction, a difference is produced and placed in accumulator B. With CMPB, the flags are set as if a difference were produced, but the difference is not retained.
19. Complement, AND, inclusive OR, and exclusive OR.
20. The upper half of the index register is loaded from the specified memory location; the lower half from the byte following the specified memory location.
21. BGE, BGT, BLE, BLT.
22. BGT is used to test the result of subtracting two's complement numbers. BHI is used to test the result of subtracting unsigned numbers.
23. NOP.
24. WAI.
25. C, I, and V.
26. Appendix A of this course.

## NEW ADDRESSING MODES

In previous units, we have discussed four addressing modes. Let's briefly review these.

In the immediate addressing mode, the operand is the memory byte immediately following the opcode. These are generally two byte instructions. The first byte is the opcode, the second is the operand. However, there are exceptions to the two-byte rule. Some operations involve the 16-bit index register and stack pointer. In these cases, the operand is the **two** bytes immediately following the opcode. These are three byte instructions. The first byte is the opcode, the second and third are the operand.

In the direct addressing mode, the byte following the opcode is the address of the operand. These are always two byte instructions. The first byte is the opcode; the second is the address of the operand. An eight-bit byte can specify addresses from 00 to FF<sub>16</sub>. Thus, when the direct addressing mode is being used, the operand must be in the first 256<sub>10</sub> bytes of memory. Since the 6800 MPU can have up to 65,536<sub>10</sub> bytes of memory, another means must be used to address the upper portion of memory.

The relative addressing mode is used for branching. These are two byte instructions. The first byte is the opcode, the second is the relative address. Recall that the relative address is added to the program count to form the absolute address. Since the 8-bit relative address is a two's complement number, the branch limits are +127<sub>10</sub> and -128<sub>10</sub>.

In the inherent addressing mode either there is no operand or the operand is implied by the instruction. These are one byte instructions.

In this section, we will discuss two new addressing modes. These are called **extended addressing** and **indexed addressing**. We will discuss extended addressing first.

## Extended Addressing

Extended addressing is similar to direct addressing but with one significant difference. Recall that with direct addressing the operand must be in the first  $256_{10}$  bytes of memory. Since this represents less than one percent of the addresses available to the 6800 MPU, a more powerful addressing mode is needed. The extended addressing mode fills this need.

The format of an instruction that uses extended addressing is shown in Figure 5-19. The instruction will always have three bytes. The first byte is the opcode. The second and third bytes form a 16-bit address. Notice that the most significant part of the address is the byte immediately following the opcode. Since this instruction has a 16-bit address, the operand can be at any one of the  $65,536_{10}$  possible addresses.

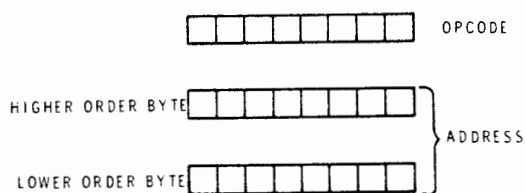


Figure 5-19  
 Format of an instruction that uses the  
 extended addressing mode.

Suppose, for example, that you wish to load the operand at memory location  $2134_{16}$  into accumulator B. The instruction would look like this:

F6	Opcode for LDAB extended
21	Higher order address
34	Lower order address

By the same token, if you wish to increment the number in memory location  $AA00_{16}$ , the instruction would be:

7C	Opcode for INC extended
AA	Higher order address
00	Lower order address

The extended addressing mode allows us to address an operand at any address including the first 256<sub>10</sub> bytes of memory. Thus, if you wish to load the operand at address 0013<sub>16</sub> into accumulator A, you can use extended addressing:

B6	Opcode for LDAA extended
00	Higher order address
13	Lower order address

Or, you can use direct addressing:

96	Opcode for LDAA direct
13	Address

Notice that, with direct addressing, the higher order address can be ignored since it is always 00. Because it saves one memory byte and one MPU cycle, direct addressing is normally used when the operand is in the first 256<sub>10</sub> bytes of memory. Extended addressing is used when the operand is above address 00FF<sub>16</sub>. However, as you will see later, some instructions do not have a direct addressing mode. In these cases, extended addressing must be used even if the operand is in the first 256<sub>10</sub> memory locations.

## Indexed Addressing

The most powerful mode available to the 6800 MPU is indexed addressing. Recall that the 6800 MPU has a 16-bit index register. There are several instructions associated with this register. They allow us to load the register from memory and to store its contents in memory. Also, we can increment and decrement the index register. We can even compare its contents with two consecutive bytes in memory. These capabilities alone make the index register a very handy 16-bit counter. However, the real power of the index register comes from the fact that we can use this counter as an address pointer. Since this is a 16-bit register, it can point to any address in memory.

**Purpose** Before going into the details of how indexed addressing works, let's see why it is needed. Let's assume that we wish to add a list of  $20_{16}$  numbers, and that the numbers are in  $20_{16}$  consecutive memory locations starting at address 0050. Using the addressing modes discussed earlier, our program might look like this:

CLRA	Clear Accumulator A.
ADDA	Add the first number
50	To accumulator A.
ADDA	Add the second
51	number to accumulator A.
ADDA	Add the third number
52	to accumulator A.
.	.
.	.
.	.
ADDA	Add the last number
6F	to accumulator A.
WAI	Wait.

While this accomplishes the desired result, it requires a long repetitive program. The above program would require  $66_{10}$  bytes of memory. Notice that all the ADDA instructions are identical except that each successive address is one larger than the previous address. Indexed addressing can greatly simplify programs of this type.

**Instruction Format** The format of an instruction that uses indexed addressing is shown in Figure 5-20. Notice that this is a two-byte instruction. The first byte is the opcode, and the second is called an offset address. The offset address is an **unsigned** 8-bit binary number. It is added to the contents of the index register to determine the address at which the operand is located.

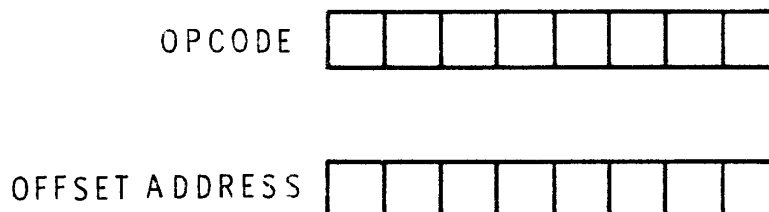


Figure 5-20

Format of an instruction that uses  
the indexed addressing mode.

Every instruction that involves an operand in memory can use the indexed addressing mode. In this unit, we will use the following convention to indicate indexed addressing:

LDAA, X  
STAA, X  
ADDB, X  
etc.

In each case, the X tells us that indexed addressing is used. For example, the first instruction means: "using indexed addressing, load the contents of the specified memory location into accumulator A." Now let's see how the address of the operand is determined.



**Determining the Operand Address** When indexed addressing is being used, the address of the operand is determined by the offset address and the number in the index register. Specifically, the 8-bit offset address is added to the 16-bit address in the index register. The 16-bit sum becomes the address of the operand. Figure 5-21 illustrates this.

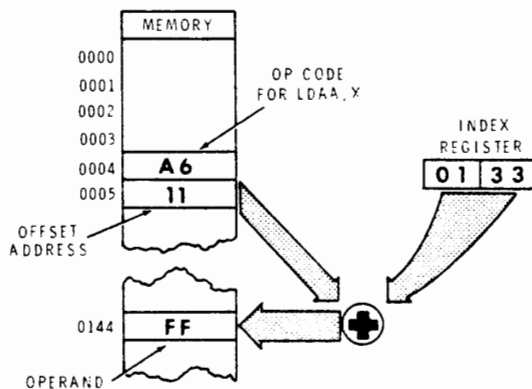


Figure 5-21

The operand address is formed by adding the offset address to the contents of the index register.

Here, the instruction in memory location  $0004_{16}$  is LDAA, X. The offset address is  $11_{16}$ . The contents of the index register are  $0133_{16}$ . When the LDAA, X instruction is executed, the address of the operand is formed by adding the offset address to the number in the index register. In this case, the operand address will be:

$$\begin{array}{r} 0133_{16} \\ + 11_{16} \\ \hline 0144_{16} \end{array}$$

The operand at this address is loaded into accumulator A. In this example, the operand FF is loaded into accumulator A when the instruction at location 0004 is executed. It is important to remember that this does not change the contents of the index register in any way. That is, the index register will still contain  $0133_{16}$  after the instruction is executed.

**Adding a List of Numbers** To see how this addressing mode saves instructions, consider the problem given earlier. Recall that we were to add  $20_{16}$  numbers stored in consecutive memory locations starting at address 0050. Using indexed addressing for the add instruction, our program looks like the one shown in Figure 5-22.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ HEX CONTENTS	COMMENTS
0010	CE	LDX #	Load index register immediate with the
0011	00	00	address of the
0012	50	50	first number in list.
0013	4F	CLRA	Clear accumulator A
0014	AB	ADDA, X	Add to accumulator A using indexed addressing
0015	00	00	with an offset address of 00.
0016	08	INX	Increment index register.
0017	8C	CPX #	Compare the contents of the index register
0018	00	00	with an address that is one greater than the
0019	70	70	address of the last number in the list.
001A	26	BNE	If not equal, branch back
001B	F8	F8	to the ADDA, X instruction.
001C	3E	WAI	Otherwise, halt.

Figure 5-22  
Program for adding a list of  
 $20_{16}$  numbers.

The first instruction is load index register immediate. Notice that a new symbol is used in this program. The symbol # is used to indicate the immediate addressing mode. Thus, the LDX# instruction causes the operand immediately following the opcode to be loaded into the index register. Recall that the index register can hold two 8-bit bytes. The operand is the two-byte number  $0050_{16}$ . You may recognize that this is the address of the first number in the list of numbers that is to be added.

The next instruction clears accumulator A. The sum will be accumulated in this register, so it is important that it be cleared initially.

The third instruction (ADDA, X) is the only instruction in the program that uses indexed addressing. Notice that the symbol X indicates the indexed addressing mode. The offset address is 00. Recall that the operand address is determined by adding the offset to the contents of the index register. The index register contains  $0050_{16}$  from a previous instruction. Since the offset is 00, the operand address is  $0050_{16}$ . That is, the contents of memory location 0050 are added to the contents of accumulator A. Recall that  $0050_{16}$  is the address of the first number in the list.

The fourth instruction increments the index register to  $0051_{16}$ . Notice that the index register now points to the address of the second number in the list.

The fifth instruction compares the number in the index register with a number that is one greater than the address of the last number in the list.

If a match occurs, the Z flag will be set. Of course in this case, no match occurs yet. Notice once again that the symbol # indicates the immediate addressing mode. Thus, the contents of the index register are compared with the next two bytes in the program or  $0070$ .

The BNE instruction tests the Z flag to see if the two numbers matched. If no match is indicated, the relative address (F8) directs the program back to the ADDA, X instruction. The first pass through the loop ends with the first number in accumulator A.

The second pass through the loop begins with the ADDA, X instruction being executed again. This time the index register points to address  $0051$ . Therefore, the second number in the list is added to accumulator A. Accumulator A now contains the sum of the first two numbers. The index register is then incremented to  $0052$ . Its contents are again compared with  $0070$ . No match exists so the BNE instruction causes the loop to be repeated again.

The loop is repeated over and over again. Each time, the next number in the list is added to accumulator A. This process continues until the last number in the list is added. At that time, the index register will be incremented to  $0070$ . Thus, when the CPX# instruction is executed, the Z flag will be set because the two numbers match. The BNE instruction recognizes that a match has occurred. Consequently, it does not allow the branch to occur and the next instruction in sequence is executed. Because this is the WAI instruction, the program halts. At this time, the sum of the  $20_{16}$  numbers in the list will be in accumulator A.

Adding a list of numbers is a classic example of how indexing can be used to shorten a program. However, this example does not illustrate the full power of indexed addressing. For example, it does not illustrate the advantage of the offset address. Because indexed addressing is so important, let's look at another example.

**Copying a List** Let's assume we have a list of  $10_{16}$  numbers that we wish to copy from one location to another. For simplicity, assume that the list is presently in addresses 0030 through 003F and that we wish to copy the list in location 0040 through 004F. Without using indexed addressing, our program might look like this:

```
LDAA
 30
STAA
 40
LDAA
 31
STAA
 41
.
.
.
LDAA
 3F
STAA
 4F
WAI
```

As you have seen; long, repetitive programs such as this are excellent candidates for indexed addressing.

Using indexed addressing, our program might look like that shown in Figure 5-23. The first step is to load the index register with the first address in the original list. The LDAA, X instruction has an offset address of 00. Therefore, accumulator A is loaded from the address specified by the index register (0030). That is, the first number in the original list is loaded into accumulator A when the LDAA, X instruction is executed.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ HEX CONTENTS	COMMENTS
0010	CE	LDX #	Load index register immediate with
0011	00	00	the first address of the original
0012	30	30	list.
0013	A6	LDAA, X	Load accumulator A indexed with
0014	00	00	an offset of 00.
0015	A7	STAA, X	Store accumulator A indexed with
0016	10	10	an offset of 10 <sub>16</sub> .
0017	08	INX	Increment index register.
0018	8C	CPX #	Compare index with one greater
0019	00	00	than last
001A	40	40	address in original list.
001B	26	BNE	If not equal, branch back to the
001C	F6	F6	LDAA, X instruction.
001D	3E	WAI	Otherwise, halt.

Figure 5-23  
 Program for copying a list from  
 addresses 0030 — 003F into  
 addresses 0040 — 004F.

The STAA, X instruction illustrates the use of the offset address. Notice that the offset is 10. This number is added to the address in the index register to form the effective address at which the contents of accumulator A are stored. Thus, the contents of accumulator A are stored at address 0040. Remember, this does not change the number in the index register in any way. By using the offset, we can load the accumulator indexed from one address and store the accumulator indexed at another.

Next, the index register is incremented to 0031. It is then compared with 0040. Since no match exists, the BNE instruction directs the program back to the LDAA, X instruction. The loop is repeated until the entire list is rewritten in locations 0040 through 004F. After the last entry in the list is copied, the index register is incremented to 0040. Thus, the CPX# instruction sets the Z flag allowing the BNE instruction to divert the program from the loop. The program halts after the last entry in the list is written in its new position in memory.

## Instruction Set Summary

You have now been introduced to most of the instructions available to the 6800 MPU. You have also been introduced to all of the addressing modes. Now let's look at the complete instruction set.

Figure 5-24 summarizes the 6800's instructions and addressing modes. This 2-page Figure contains a wealth of information. For your convenience, this information is repeated on the Instruction Set Summary card provided with the course. You should keep this card handy. After a while, you will be able to write long, complex programs using only the card for reference.

The left-hand column of Figure 5-24 lists the names and mnemonics for each of the instructions. In many cases, a single name such as "add" is associated with more than one mnemonic. For example, ADDA is an add operation that involves accumulator A while ADDB is an add operation that involves accumulator B.

The center column gives important information about the addressing modes. Notice that the ADDA instruction can have any one of four addressing modes: immediate, direct, indexed, or extended. Three facts are given for each addressing mode. The hexadecimal opcode is given in the OP column. For example, the opcode for ADDA immediate is 8B while the opcode for ADDA direct is 9B.

The column labeled (~) tells the number of MPU cycles required to execute the instruction. This information is important because it allows us to determine exactly how long it will take to run a given program. As you will see later, an MPU cycle is equal to one cycle of the MPU clock. For example, if the clock frequency is 1 MHz, one MPU cycle will be one microsecond. With this clock rate, 2 microseconds are required to execute the ADDA immediate instruction while 5 microseconds are required for the ADDA indexed instruction.

The column labeled (#) indicates the number of bytes required by the instruction. ADDA immediate, ADDA direct, and ADDA indexed are two-byte instructions while ADDA extended is a three-byte instruction.

The next column to the right gives the shorthand notation for the Boolean or arithmetic operations performed. Finally, the right-hand column indicates how the condition code registers are affected by each instruction.

If you study the instruction set carefully, you will find that there are a few instructions that we have not yet discussed. These will be described in the next unit.

ACCUMULATOR AND MEMORY		ADDRESSING MODES															BOOLEAN/ARITHMETIC OPERATION (All register labels refer to contents)		COND. CODE REG.					
		IMMED			DIRECT			INDEX			EXTND			INNER					5	4	3	2	1	0
		OP	~	#	OP	~	#	OP	~	#	OP	~	#	OP	~	#			N	I	Z	V	C	
OPERATIONS	MNEMONIC																							
Add	ADDA	88	2	2	98	3	2	A8	5	2	B8	4	3				A ← M → A		1	0	1	1	1	
	ADDB	CB	2	2	D8	3	2	E8	5	2	F8	4	3				B ← M → B		1	0	1	1	1	
Add Acmltrs	ABA													1B	2	1	A ← B → A		1	0	1	1	1	
Add with Carry	ADCA	89	2	2	99	3	2	A9	5	2	B9	4	3				A ← M + C → A		1	0	1	1	1	
	ADCB	C9	2	2	D9	3	2	E9	5	2	F9	4	3				B ← M + C → B		1	0	1	1	1	
And	ANDA	84	2	2	94	3	2	A4	5	2	B4	4	3				A ← M → A		0	0	1	1	1	
	ANDB	C4	2	2	D4	3	2	E4	5	2	F4	4	3				B ← M → B		0	0	1	1	1	
Bit Test	BITA	85	2	2	95	3	2	A5	5	2	B5	4	3				A ← M		0	0	1	1	1	
	BITB	C5	2	2	D5	3	2	E5	5	2	F5	4	3				B ← M		0	0	1	1	1	
Clear	CLR							6F	7	2	7F	6	3				00 ← M		0	0	1	1	1	
	CLRA													4F	2	1	00 → A		0	0	1	1	1	
	CLRB													5F	2	1	00 → B		0	0	1	1	1	
Compare	CMPA	81	2	2	91	3	2	A1	5	2	B1	4	3				A ← M		0	0	1	1	1	
	CMPB	C1	2	2	D1	3	2	E1	5	2	F1	4	3				B ← M		0	0	1	1	1	
Compare Acmltrs	CBA													11	2	1	A ← B		0	0	1	1	1	
Complement, 1's	COM							63	7	2	73	6	3				M ← M		0	0	1	1	1	
	COMA													43	2	1	A ← A		0	0	1	1	1	
	COMB													53	2	1	B ← B		0	0	1	1	1	
Complement, 2's (Negate)	NEG							60	7	2	70	6	3				00 ← M → M		0	0	1	1	1	
	NEGA													40	2	1	00 ← A → A		0	0	1	1	1	
	NEGB													50	2	1	00 ← B → B		0	0	1	1	1	
Decimal Adjust, A	DAA													19	2	1	Converts Binary Add. of BCD Characters into BCD Format		0	0	1	1	1	
Decrement	DEC							6A	7	2	7A	6	3				M ← 1 → M		0	0	1	1	1	
	DECA													4A	2	1	A ← 1 → A		0	0	1	1	1	
	DECB													5A	2	1	B ← 1 → B		0	0	1	1	1	
Exclusive OR	EORA	88	2	2	98	3	2	A8	5	2	B8	4	3				A ← M → A		0	0	1	1	1	
	EORB	C8	2	2	D8	3	2	E8	5	2	F8	4	3				B ← M → B		0	0	1	1	1	
Increment	INC							6C	7	2	7C	6	3				M ← 1 → M		0	0	1	1	1	
	INCA													4C	2	1	A ← 1 → A		0	0	1	1	1	
	INCB													5C	2	1	B ← 1 → B		0	0	1	1	1	
Load Acmltr	LDAA	86	2	2	96	3	2	A6	5	2	B6	4	3				M → A		0	0	1	1	1	
	LDAB	C6	2	2	D6	3	2	E6	5	2	F6	4	3				M → B		0	0	1	1	1	
Or, Inclusive	ORAA	8A	2	2	9A	3	2	AA	5	2	BA	4	3				A ← M → A		0	0	1	1	1	
	ORAB	CA	2	2	DA	3	2	EA	5	2	FA	4	3				B ← M → B		0	0	1	1	1	
Push Data	PSHA													36	4	1	A → Msp, SP ← 1 → SP		0	0	1	1	1	
	PSHB													37	4	1	B → Msp, SP ← 1 → SP		0	0	1	1	1	
Pull Data	PULA													32	4	1	SP ← 1 → SP, Msp → A		0	0	1	1	1	
	PULB													33	4	1	SP ← 1 → SP, Msp → B		0	0	1	1	1	
Rotate Left	ROL							69	7	2	79	6	3				M		0	0	1	1	1	
	ROLA													49	2	1	A		0	0	1	1	1	
	ROLB													59	2	1	B		0	0	1	1	1	
Rotate Right	ROR							66	7	2	76	6	3				M		0	0	1	1	1	
	RORA													46	2	1	A		0	0	1	1	1	
	RORB													56	2	1	B		0	0	1	1	1	
Shift Left, Arithmetic	ASL							68	7	2	78	6	3				M		0	0	1	1	1	
	ASLA													48	2	1	A		0	0	1	1	1	
	ASLB													58	2	1	B		0	0	1	1	1	
Shift Right, Arithmetic	ASR							67	7	2	77	6	3				M		0	0	1	1	1	
	ASRA													47	2	1	A		0	0	1	1	1	
	ASRB													57	2	1	B		0	0	1	1	1	
Shift Right, Logic	LSR							64	7	2	74	6	3				M		0	0	1	1	1	
	LSRA													44	2	1	A		0	0	1	1	1	
	LSRB													54	2	1	B		0	0	1	1	1	
Store Acmltr	STAA				97	4	2	A7	6	2	B7	5	3				A → M		0	0	1	1	1	
	STAB				07	4	2	E7	6	2	F7	5	3				B → M		0	0	1	1	1	
Subtract	SUBA	80	2	2	90	3	2	A0	5	2	B0	4	3				A ← M → A		0	0	1	1	1	
	SUBB	C0	2	2	D0	3	2	E0	5	2	F0	4	3				B ← M → B		0	0	1	1	1	
Subtract Acmltrs	SBA													10	2	1	A ← B → A		0	0	1	1	1	
Subtr. with Carry	SBCA	82	2	2	92	3	2	A2	5	2	B2	4	3				A ← M - C → A		0	0	1	1	1	
	SBCB	C2	2	2	D2	3	2	E2	5	2	F2	4	3				B ← M - C → B		0	0	1	1	1	
Transfer Acmltrs	TAB													16	2	1	A → B		0	0	1	1	1	
	TBA													17	2	1	B → A		0	0	1	1	1	
Test, Zero or Minus	TST							6D	7	2	7D	6	3				M ← 00		0	0	1	1	1	
	TSTA													4D	2	1	A ← 00		0	0	1	1	1	
	TSTB													5D	2	1	B ← 00		0	0	1	1	1	

Figure 5-24

The 6800 instruction set.

INDEX REGISTER AND STACK		IMMED			DIRECT			INDEX			EXTND			INNER			BOOLEAN/ARITHMETIC OPERATION						
POINTER OPERATIONS	MNEMONIC	OP	~	#	OP	~	#	OP	~	#	OP	~	#	OP	~	#		H	I	N	Z	V	C
Compare Index Reg	CPX	8C	3	3	9C	4	2	AC	6	2	6C	5	3				$(X_H/X_L) - (M/M + 1)$	•	•	⑦	†	⑧	•
Decrement Index Reg	DEX													09	4	1	$X - 1 \rightarrow X$	•	•	•	†	•	•
Decrement Stack Pntr	DES													34	4	1	$SP - 1 \rightarrow SP$	•	•	•	•	•	•
Increment Index Reg	INX													28	4	1	$X + 1 \rightarrow X$	•	•	•	†	•	•
Increment Stack Pntr	INS													31	4	1	$SP + 1 \rightarrow SP$	•	•	•	•	•	•
Load Index Reg	LIX	CE	3	3	DE	4	2	EE	6	2	FE	5	3				$M \rightarrow X_H, (M + 1) \rightarrow X_L$	•	•	⑨	†	R	•
Load Stack Pntr	LDS	BE	3	3	9E	4	2	AE	6	2	BE	5	3				$M \rightarrow SP_H, (M + 1) \rightarrow SP_L$	•	•	⑨	†	R	•
Store Index Reg	STX				0F	5	2	EF	7	2	FF	6	3				$X_H \rightarrow M, X_L \rightarrow (M + 1)$	•	•	⑨	†	R	•
Store Stack Pntr	STS				9F	5	2	AF	7	2	BF	6	3				$SP_H \rightarrow M, SP_L \rightarrow (M + 1)$	•	•	⑨	†	R	•
Idx Reg → Stack Pntr	TXS													35	4	1	$X - 1 \rightarrow SP$	•	•	•	•	•	•
Stack Pntr → Idx Reg	TSX													3C	4	1	$SP + 1 \rightarrow X$	•	•	•	•	•	•

JUMP AND BRANCH		RELATIVE			INDEX			EXTND			INNER			BRANCH TEST								
OPERATIONS	MNEMONIC	OP	~	#	OP	~	#	OP	~	#	OP	~	#		H	I	N	Z	V	C		
Branch Always	BRA	20	4	2										None	•	•	•	•	•	•		
Branch If Carry Clear	BCC	24	4	2										$C = 0$	•	•	•	•	•	•		
Branch If Carry Set	BCS	25	4	2										$C = 1$	•	•	•	•	•	•		
Branch If = Zero	BEQ	27	4	2										$Z = 1$	•	•	•	•	•	•		
Branch If ≥ Zero	BGE	2C	4	2										$N \div V = 0$	•	•	•	•	•	•		
Branch If > Zero	BGT	2E	4	2										$Z + (N \div V) = 0$	•	•	•	•	•	•		
Branch If Higher	BHI	22	4	2										$C + Z = 0$	•	•	•	•	•	•		
Branch If ≤ Zero	BLE	2F	4	2										$Z + (N \div V) = 1$	•	•	•	•	•	•		
Branch If Lower Or Same	BLS	23	4	2										$C + Z = 1$	•	•	•	•	•	•		
Branch If < Zero	BLT	2D	4	2										$N \div V = 1$	•	•	•	•	•	•		
Branch If Minus	BMI	2B	4	2										$N = 1$	•	•	•	•	•	•		
Branch If Not Equal Zero	BNE	26	4	2										$Z = 0$	•	•	•	•	•	•		
Branch If Overflow Clear	BVC	28	4	2										$V = 0$	•	•	•	•	•	•		
Branch If Overflow Set	BVS	29	4	2										$V = 1$	•	•	•	•	•	•		
Branch If Plus	BPL	2A	4	2										$N = 0$	•	•	•	•	•	•		
Branch To Subroutine	BSR	8D	8	2											•	•	•	•	•	•		
Jump	JMP				6E	4	2	7E	3	3				See Special Operations	•	•	•	•	•	•		
Jump To Subroutine	JSR				AD	8	2	BD	9	3					•	•	•	•	•	•	•	
No Operation	NOP												01	2	1	Advances Prog. Cntr. Only	•	•	•	•	•	•
Return From Interrupt	RTI												3B	10	1		•	•	•	•	•	•
Return From Subroutine	RTS												39	5	1	See special Operations	•	•	•	•	•	•
Software Interrupt	SWI												3F	12	1		•	•	•	•	•	•
Wait for Interrupt	WAI												3E	9	1	⑩	•	•	•	•	•	•

CONDITIONS CODE REGISTER		INNER			BOOLEAN OPERATION			CONDITION CODE REGISTER NOTES:						
OPERATIONS	MNEMONIC	OP	~	#	OP	~	#		H	I	N	Z	V	C
Clear Carry	CLC	0C	2	1	0 → C			① (Bit V) Test: Result = 10000000?	•	•	•	•	•	P
Clear Interrupt Mask	CLI	0E	2	1	0 → I			② (Bit C) Test: Result = 00000000?	•	R	•	•	•	•
Clear Overflow	CLV	0A	2	1	0 → V			③ (Bit C) Test: Decimal value of most significant BCD Character greater than nine? (Not cleared if previously set.)	•	•	•	•	R	•
Set Carry	SEC	0D	2	1	1 → C				•	•	•	•	•	S
Set Interrupt Mask	SEI	0F	2	1	1 → I				•	S	•	•	•	•
Set Overflow	SEV	0B	2	1	1 → V				•	•	•	•	S	•
Accmtr A → CCR	TAP	06	2	1	A → CCR				•	•	•	•	•	•
CCR → Accmtr A	TPA	07	2	1	CCR → A				•	•	•	•	•	•

## LEGEND:

OP	Operation Code (Hexadecimal).	H	Half-carry from bit 3;
~	Number of MPU Cycles;	I	Interrupt mask
=	Number of Program Bytes;	N	Negative (sign bit)
+	Arithmetic Plus;	Z	Zero (byte)
-	Arithmetic Minus;	V	Overflow, 2's complement
•	Boolean AND;	C	Carry from bit 7
M <sub>SP</sub>	Contents of memory location pointed to be Stack Pointer;	R	Reset Always
+	Boolean Inclusive OR;	S	Set Always
÷	Boolean Exclusive OR;	†	Test and set if true, cleared otherwise
M̄	Complement of M;	•	Not Affected
→	Transfer Into;	CCR	Condition Code Register
0	Bit = Zero;	LS	Least Significant
		MS	Most Significant

 Figure 5-24  
 (continued)



## Self-Test Review

27. A disadvantage of direct addressing is that the operand must be in the first \_\_\_\_\_ bytes of memory.
28. The advantage of direct addressing is that only \_\_\_\_\_ bytes are required for each instruction.
29. Extended addressing can address \_\_\_\_\_ bytes of memory.
30. A disadvantage of extended addressing is that each instruction requires \_\_\_\_\_ bytes.
31. Can extended addressing be used to address an operand in the first 256<sub>10</sub> bytes of memory?
32. The most powerful addressing mode available to the 6800 is called \_\_\_\_\_ addressing.
33. Indexed addressing requires \_\_\_\_\_ bytes for each instruction.
34. The second byte of an indexed addressing instruction is called the \_\_\_\_\_ address.
35. How is the address of the operand determined when indexed addressing is used?
36. Carefully examine the program shown in Figure 5-25. Determine what the program does and fill in the comments column. What number is loaded into the index register by the first instruction?

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ HEX CONTENTS	COMMENTS
0010	CE	LDX #	
0011	00	00	
0012	50	50	
0013	6F	CLR. X	
0014	00	00	
0015	08	INX	
0016	8C	CPX #	
0017	00	00	
0018	60	60	
0019	26	BNE	
001A	F8	F8	
001B	3E	WAI	

Figure 5-25  
Program for Self-Test Review

37. What location is cleared by the CLR, X instruction?
38. What is the number in the index register after the INX instruction is executed for the first time?
39. The loop will be repeated until the number in the index register is \_\_\_\_\_.
40. What does this program do?
41. Refer to Figure 5-24. What is the hexadecimal opcode for the LDAB extended instruction?
42. How many MPU cycles are required by the INC, X instruction?
43. How many bytes in the LDS # instruction?

## Answers

- 27.  $256_{10}$ .
- 28. Two.
- 29.  $65,536_{10}$ .
- 30. Three.
- 31. Yes. Although direct addressing is normally used when the operand is in the first  $256_{10}$  bytes of memory, extended addressing can be used also.
- 32. Indexed.
- 33. Two.
- 34. Offset.
- 35. The offset address is added to the contents of the index register.
- 36.  $0050_{16}$ .
- 37.  $0050_{16}$ .
- 38.  $0051_{16}$ .
- 39.  $0060_{16}$ .
- 40. The program clears memory locations  $0050_{16}$  through  $005F_{16}$ .
- 41.  $F6_{16}$ .
- 42. Seven.
- 43. Three.

## EXPERIMENTS

Perform Programming Experiments 7 and 8. You will find these experiments in Unit 9. After you finish these experiments, return to this unit and complete the Unit Examination.

## UNIT EXAMINATION

1. Which of the following program segments will **not** clear both accumulators?
  - A. CLRA  
CLRB
  - B. CLRA  
TAB
  - C. CLRB  
TBA
  - D. CLRA  
ABA
2. Which of the following contains an operation that can **not** be performed directly on a byte in memory using a single instruction?
  - A. Increment, decrement, shift left arithmetically.
  - B. Clear, complement, compare.
  - C. Rotate left, negate, test for zero.
  - D. Shift right logically, rotate right, test for minus.
3. Which addressing mode is best suited for adding a list of numbers?
  - A. Direct.
  - B. Extended.
  - C. Indexed.
  - D. Relative.

4. Which of the following program segments will successfully swap the contents of accumulators A and B?

- |         |         |
|---------|---------|
| A. TAB  | C. TAB  |
| TBA     | ABA     |
| B. STAA | D. STAA |
| 10      | 10      |
| TBA     | LDAB    |
| LDAB    | 10      |
| 10      | TBA     |

5. Which of the following program segments will cause a branch if the number in memory location 8310 is odd?

- |        |         |
|--------|---------|
| A. ROR | C. RORA |
| 83     | BCS     |
| 10     | 07      |
| BCS    |         |
| 07     | D. LDAA |
|        | 83      |
| B. ASL | 10      |
| 83     | ROLA    |
| 10     | BCS     |
| BCS    | 07      |
| 07     |         |

6. Examine the following program segment:

```
CLRA
INCA
BNE
FD
WAI
```

If an MPU cycle is 1 microsecond, how much time elapses from the time this segment starts running until the WAI instruction is fetched?

- A. Approximately 8 microseconds.  
B. Approximately 2050 microseconds.  
C. Approximately 1538 microseconds.  
D. Approximately 3 microseconds.

7. Which of the following instructions can be used to clear the Z flag?

- A. BEQ
- B. BNE
- C. NOP
- D. TAP

8. Which of the following instructions can be used to test the result of the subtraction of unsigned binary numbers?

- A. BGE.
- B. BGT.
- C. BCS.
- D. BLT

9. Examine the following program segment:

```
LDX #  
    00  
    50  
    → DEX  
    BNE  
    FD  
    WAI
```

How many times will the DEX instruction be executed?

- A. Once
- B.  $50_{16}$  times.
- C.  $65,536_{10}$ .
- D. The number of times will depend on the contents of memory location 0050.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ HEX CONTENTS	COMMENTS
0010	4F	CLRA	Clear Accumulator A.
0011	7D	TST	Test
0012	00	00	the
0013	1E	1E	multiplier.
0014	27	BEQ	If it is zero branch to wait.
0015	07	07	
0016	7A	DEC	Otherwise decrement
0017	00	00	the
0018	1E	1E	multiplier.
0019	9B	ADDA	Add the
001A	1F	1F	multiplier to the product.
001B	20	BRA	Repeat the loop.
001C	F4	F4	
001D	3E	WAI	Wait.
001E	05	Multiplier	
001F	04	Multiplcand	

Figure 5-26

This program multiplies by repeated addition.

NOTE: Refer to the program shown in Figure 5-26 for questions 10 through 16.

10. What addressing mode does the TST instruction use?
  - A. Immediate
  - B. Direct.
  - C. Extended.
  - D. Indexed.
11. The BEQ instruction checks to see if the TST instruction set the:
  - A. Z flag
  - B. C flag.
  - C. H flag.
  - D. V flag.
12. The DEC instruction decrements the number in:
  - A. Accumulator A.
  - B. Memory location 001E.
  - C. Accumulator B.
  - D. The index register.



13. Which instruction is executed immediately after the BRA instruction?
- A. WAI.
  - B. BEQ.
  - C. CLRA.
  - D. TST.
14. With the values given for the multiplier and multiplicand, how many times will the main program loop be repeated?
- A. Four times.
  - B. Five times.
  - C. Twenty times.
  - D. Twice.
15. After the program has been executed, memory location 001E will contain:
- A.  $05_{16}$ .
  - B.  $04_{16}$ .
  - C.  $20_{16}$ .
  - D.  $00_{16}$ .
16. After the program has been executed, the product will appear in:
- A. Memory location 001E.
  - B. Memory location 001F.
  - C. Accumulator A.
  - D. Accumulator B.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ HEX CONTENTS	COMMENTS
0010	CE	LDX #	
0011	00	00	
0012	00	05	
0013	A6	→ LDAA, X	
0014	20	20	
0015	AB	ADDA, X	
0016	30	30	
0017	A7	STAA, X	
0018	40	40	
0019	08	INX	
001A	8C	CPX #	
001B	00	00	
001C	15	15	
001D	26	BNE	
001E	F4	F4	
001F	3E	WAI	

Figure 5-27

Program for Questions 17 through 20.

NOTE: Refer to Figure 5-27 for Questions 17 through 20. Analyze the program, determine what it does, and fill in appropriate comments.

17. On the first pass through the main program loop, the LDAA, X instruction takes its operand from memory location:

- A. 0005.
- B. 0020.
- C. 0025.
- D. 0014.

18. On the first pass, the ADDA, X adds the contents of what memory location to accumulator A?
- A. 0005.
  - B. 0030.
  - C. 0035.
  - D. 0016.
19. On the second pass through the program loop, the contents of memory location:
- A. 0021 are added to the contents of 0031 and the result is stored in 0041.
  - B. 0026 are added to the contents of 0036 and the result is stored in 0046.
  - C. 0025 are added to the contents of 0035 and the result is stored in 0045.
  - D. 0020 are added to the contents of 0030 and the result is stored in 0040.
20. How many times is the main program loop repeated?
- A.  $10_{16}$  times.
  - B.  $05_{16}$  times.
  - C.  $30_{16}$  times.
  - D.  $15_{16}$  times.





# Individual Learning Program

## MICROPROCESSORS

### *Unit 6*

## THE 6800 MICROPROCESSOR — PART 2

EE-3401

HEATH COMPANY  
BENTON HARBOR, MICHIGAN 49022

Copyright © 1977  
Heath Company  
All Rights Reserved  
Printed in the United States of America

## CONTENTS

Introduction .....	6-3
Unit Objectives .....	6-4
Unit Activity Guide .....	6-5
Stack Operations .....	6-6
Subroutines .....	6-17
Input-Output (I/O) Operations .....	6-27
Interrupts .....	6-37
Experiments .....	6-50
Unit Examination .....	6-51
Examination Answers .....	6-53

## *Unit 6*

# **THE 6800 MICROPROCESSOR — PART 2**

## **INTRODUCTION**

In the previous unit, you were introduced to the architecture and instruction set of the 6800 microprocessor. Much of the MPU's capabilities were discussed; however, three important areas were omitted. These include the microprocessor's stack operation, the use of subroutines, and the interrupt capability. These capabilities are discussed in detail in this unit. You are also introduced to input-output operations.

## UNIT OBJECTIVES

When you have completed this unit, you will be able to:

1. Explain the difference between a cascade stack and a memory stack.
2. Write simple programs that can store data in — and retrieve data from — the stack.
3. Write programs that use the stack and indexing to move a list from one place in memory to another.
4. Explain the operations performed by each of the following instructions: PULA, PULB, PSHA, PSHB, DES, INS, LDS, STS, TXS, and TSX.
5. Define stack, subroutine, nested subroutine, interrupt, interrupt vector, and interrupt masking.
6. Write programs that use subroutines and nested subroutines.
7. Explain the operations performed by each of the following instructions: JMP, JSR, BSR, and RTS.
8. Describe how the 6800 MPU performs input and output operations.
9. Draw flowcharts depicting the sequence of events that occur during reset, non-maskable interrupt, interrupt request, software interrupt, return from interrupt, and wait for interrupt.
10. Explain the operation performed by each of the following instructions: WAI, SWI, RTI, SEI, and CLI.



## UNIT ACTIVITY GUIDE

Completion  
Time

- |   |       |
|---|-------|
| <input type="checkbox"/> Read Section on Stack Operations.          | _____ |
| <input type="checkbox"/> Complete Self-Test Review Questions 1-10.  | _____ |
| <input type="checkbox"/> Read Section on Subroutines.               | _____ |
| <input type="checkbox"/> Complete Self-Test Review Questions 11-20. | _____ |
| <input type="checkbox"/> Read Section on Input-Output Operations.   | _____ |
| <input type="checkbox"/> Complete Self-Test Review Questions 21-27. | _____ |
| <input type="checkbox"/> Read Section on Interrupts.                | _____ |
| <input type="checkbox"/> Complete Self-Test Review Questions 28-40. | _____ |
| <input type="checkbox"/> Perform Programming Experiments 9 and 10.  | _____ |
| <input type="checkbox"/> Complete Unit Examination.                 | _____ |
| <input type="checkbox"/> Check Examination Answers.                 | _____ |

## STACK OPERATIONS

In computer jargon, a *stack* is a group of temporary storage locations in which data can be stored and later retrieved. In this regard, a *stack* is somewhat like memory. In fact, many microprocessors use a section of memory as a stack. The difference between a stack and other forms of memory is the method by which the data is accessed or addressed. The discussion will begin by considering a simple stack arrangement used in some microprocessors. Then the more sophisticated stack arrangement used by the 6800 MPU will be discussed.

### Cascade Stack

Some microprocessors have a special group of registers (usually 8 or 16) called a *cascade stack*. Each register can hold one 8-bit byte of data. Because these registers are right on the MPU chip, they make excellent temporary storage locations. If we need to free the accumulator for some reason, we can store its contents in the stack. Later, if that piece of data is needed again, we can retrieve the data from the stack. Of course, we could also have freed the accumulator by storing the data in memory. What then is the advantage of the stack?

One advantage of the stack is the method by which it is accessed or addressed. Recall that when a byte is stored in memory, an address is required. That is to store the contents of the accumulator in memory a 2-byte or 3-byte instruction is required. Depending on the addressing mode, the last one or two bytes is the address. Later, if the byte is retrieved, another instruction is required that also has an address.

An advantage of the stack is that data can be stored into it or read from it with single-byte instructions. That is, the instructions used with the stack do not require an address. Therefore, they are single-byte instructions.

Figure 6-1 shows an 8-register stack similar to that found in some microprocessors. This is called a cascade stack because of the method by which data is loaded and retrieved. All data transfers are between the top of the stack and the accumulator. That is, the accumulator communicates only with the top location on the stack. Data is transferred to the stack by a special instruction called PUSH.

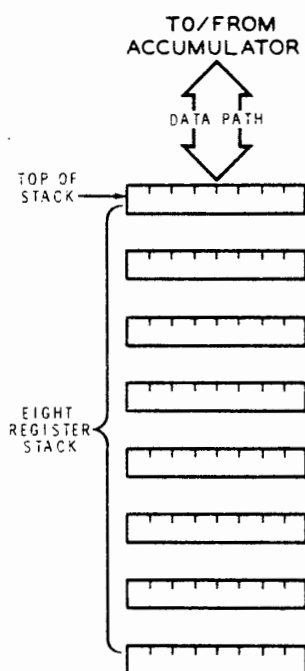


Figure 6-1.  
A cascade stack.

**The PUSH Instruction.** Figure 6-2 illustrates how the PUSH instruction places data in the stack. The number  $01_{16}$  is in the accumulator and we wish to temporarily store it. While we could store the number in memory, this would require a 2-byte or a 3-byte instruction. So instead, we use the PUSH instruction to place this number in the stack. Notice that the number is placed in the top location of the stack as shown in Figure 6-2A. The number remains there until we retrieve it or until we push another byte into the stack.

Figure 6-2B shows what happens if, at some time later, we push another byte into the stack. Notice that the accumulator now contains  $03_{16}$ . If the PUSH instruction is executed, the contents of the accumulator are pushed into the top of the stack. To make room for this new number, the original number  $01_{16}$  is pushed deeper into the stack.

Figures 6-2C and 6-2D show two more numbers being pushed into the stack at later points in the program. Notice that new data is always pushed into the top of the stack. To make room for the new data, the old data is pushed deeper into the stack. For this reason, this arrangement is often called a *push-down* or *cascade stack*. The name *cascade stack* comes from the characteristic cascading of data down through the stack as each new byte is pushed in at the top.

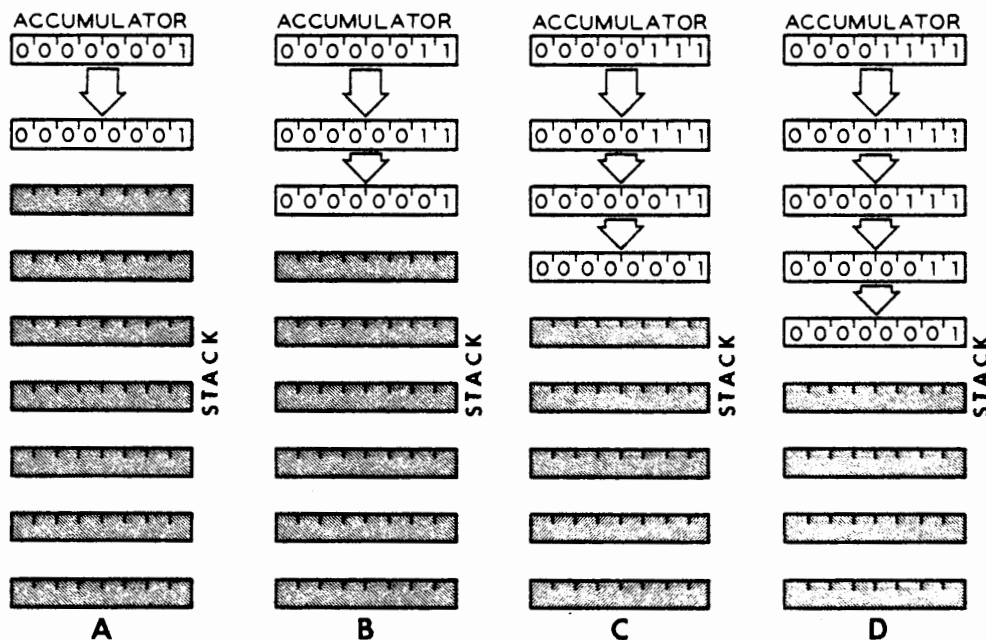


Figure 6-2.  
Pushing data into the stack.

**The PULL Instruction.** The MPU retrieves data from the stack by using the PULL instruction. In some microprocessors, this is referred to as a POP instruction.

Figure 6-3 illustrates how data can be pulled (or popped) from the stack. Figure 6-3A shows the stack as it appeared after the last push operation. Notice that it contains four bytes of data. The last byte of data that was entered is at the top of the stack.

The PULL instruction retrieves the byte that is at the top of the stack. As this byte is removed from the stack, all other bytes move up, filling in the space left by that byte. Figure 6-3B illustrates how  $0F_{16}$  is pulled from the stack. Notice that  $07_{16}$  is now at the top of the stack.

Figures 6-3C and 6-3D show how the next two bytes can be pulled from the stack. In each case, the remaining bytes move up in the stack, filling in the register vacated by the removed byte.

If you compare Figures 6-2 and 6-3, you will notice that the data must be pulled from the stack in the reverse order. That is, the last byte pushed into the stack is the first byte that is pulled from the stack. Another name for this arrangement is a last-in/first-out (LIFO) stack.

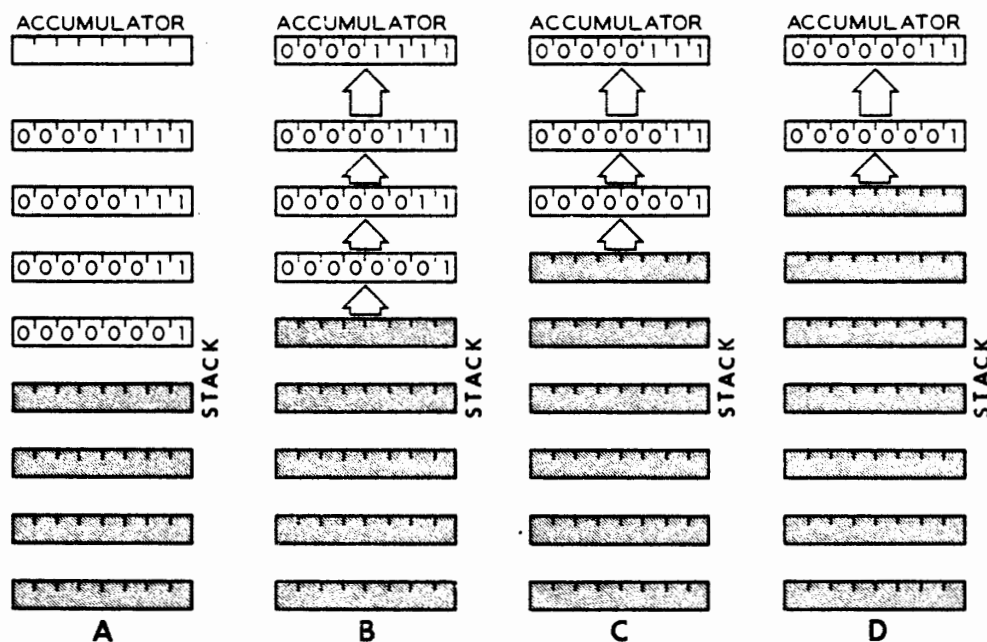


Figure 6-3.  
Pulling data from the stack.

## Memory Stack

While a cascade stack is valuable, it does have some limitations. For one thing, the number of registers is generally quite limited, with eight being typical. If more than eight pieces of data are pushed into the stack, the "older" bytes are pushed out the bottom and are lost. Also, the readout of the stack is destructive. When a byte is pulled from the stack, it no longer exists in the stack. This is fundamentally different from reading a byte from memory.

Because of these limitations the 6800 MPU does not use a cascade stack. Instead, a section of RAM can be set aside by the programmer to act as a stack. This has several advantages. First, the stack can be any length that the programmer requires. Second, the programmer can set up more than one stack if he likes. Third he can address the data in the stack using any of the instructions that address memory.

**Stack Pointer.** Recall that the 6800 MPU has a 16-bit register called the stack pointer. In a memory-type stack, the stack pointer defines the memory location that acts as the top of the stack.

The cascade stack considered earlier generally does not require a stack pointer. The top of the stack is determined by hardware. During push and pull operations, the data bytes actually move from one register to another. That is, the top of the stack remains stationary and the data moves up or down in relation to the stack.

In the memory stack, data cannot be easily transferred from one location to the next. Therefore, instead of moving data up and down in relation to the stack, it is much easier to move the top of the stack in relation to the data.

Generally, when the microprocessor-based system is being planned, a section of RAM is reserved for the stack. This should be a section of RAM that is not being used for any other purpose.

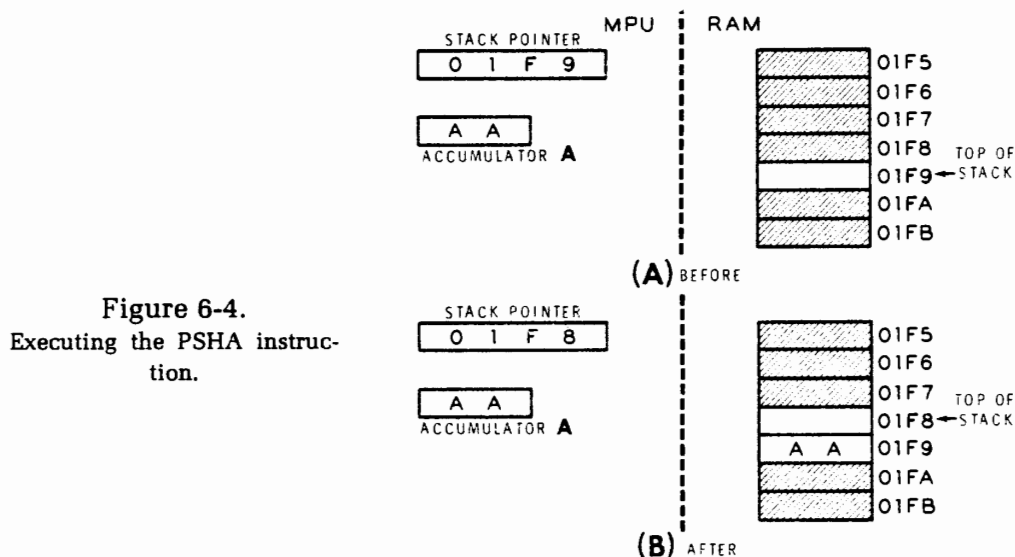
Once this is done, the stack can be set up by a program. The top of the stack is established by loading an address into the stack pointer. For example, suppose we wish to establish address 01F9<sub>16</sub> as the top of the stack. The following instruction could be used:

```
LDS#  
01  
F9
```

This loads the address  $01F9_{16}$  into the stack pointer and establishes that address as the top of the stack. However, as you will see, the top of the stack moves each time data is pushed into — or pulled from — the stack.

**The PUSH Instructions.** The 6800 MPU has two push instructions, PSHA and PSHB. These single-byte instructions push the contents of their respective accumulator onto the stack.

Figure 6-4 shows the effects of the PSHA instruction. Before the instruction is executed, the stack pointer contains the address  $01F9_{16}$  as a result of a previous LDS instruction. Accumulator A contains a data byte ( $AA_{16}$ ). If the PSHA instruction is now executed, the contents of accumulator A are pushed into memory location  $01F9_{16}$ . Then, the stack pointer is automatically decremented to  $01F8_{16}$ . This automatically moves the top of the stack as shown.



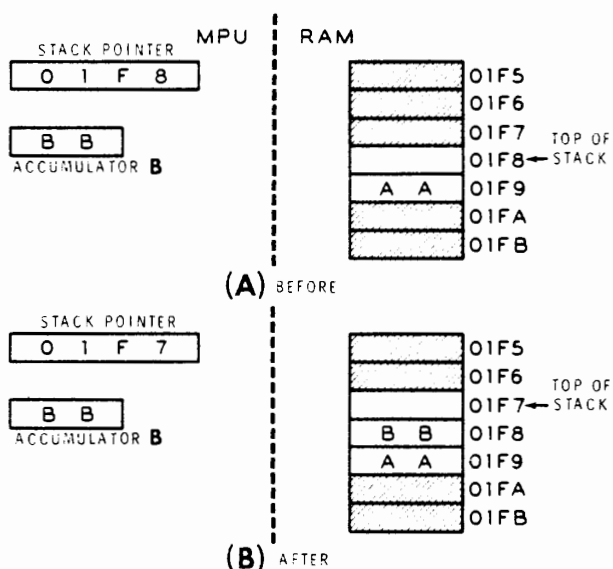
If you look at your Instruction Set Summary card, you will see that the operation is described as follows:

$$A \rightarrow M_{SP}, SP - 1 \rightarrow SP$$

This means that the contents of the A accumulator are transferred to the memory location specified by the stack pointer. Also, the contents of the stack pointer are replaced by the previous contents of the stack pointer minus one. In other words, after the accumulator-to-stack transfer takes place, the stack pointer is decremented by one.

To reinforce the idea, assume that at some later point in the program, the MPU executes a PSHB instruction. This is illustrated in Figure 6-5. Before PSHB is executed, the B accumulator contains  $BB_{16}$  and the stack pointer is still pointing to  $01F8_{16}$ . When PSHB is executed, the contents of accumulator B are pushed onto the stack and the stack pointer is decremented to  $01F7_{16}$ .

**The PULL Instructions.** Data bytes are removed from the stack with the pull instruction. The 6800 MPU has two pull instructions. PULA allows the MPU to pull data from the stack into the A accumulator. PULB performs a similar operation except the data byte goes into accumulator B. In each case, data is pulled from the top of the stack. Thus, the data byte available to the MPU is the last byte that was placed in the stack.

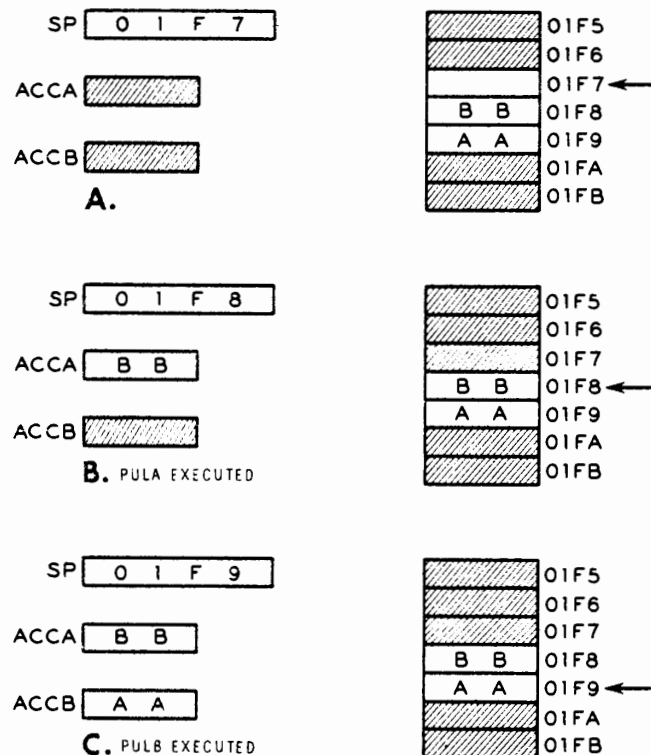


**Figure 6-5.**  
Executing the PSHB instruction.

For example, Figure 6-6A shows the stack as we left it after the last push instruction. Figure 6-6B shows what happens if the PULA instruction is executed. First, the stack pointer is automatically incremented by one to  $01F8_{16}$ . Then the contents of the memory location designated by the stack pointer are transferred to accumulator A. Thus,  $BB_{16}$  goes into accumulator A. Notice that the stack pointer is incremented before the byte is pulled from the stack.

To be certain you have the idea, consider what happens if the PULB instruction is now executed. Figure 6-6C shows that the stack pointer is automatically incremented to  $01F9_{16}$ . The contents of that location are then pulled into accumulator B. This operation is described on your Instruction Set Summary card as:

$$SP + 1 \rightarrow SP, M_{SP} \rightarrow B.$$



**Figure 6-6.**  
Executing PULL instructions.



**Using the Stack.** Figure 6-7 summarizes all of the instructions that directly affect stack operations. The push and pull instructions were introduced in this unit while the other instructions were discussed briefly in the previous unit. Find these instructions on your Instruction Set Summary card. The push and pull instructions are listed with the Accumulator and Memory Operations. Those instructions that affect the stack pointer are listed under Index Register and Stack Pointer Operations.

tions.

ADDRESSING MODES

STACK AND STACK POINTER OPERATIONS		MNEMONIC		ADDRESSING MODES															BOOLEAN/ARITHMETIC OPERATION (All register labels refer to contents)
				IMMED			DIRECT			INDEX			EXTND			INHER			
OP	~	#	OP	~	#	OP	~	=	OP	~	#	OP	~	#	OP	~	#		
Push Data	PSHA														36	4	1	A → M <sub>SP</sub> , SP - 1 → SP	
	PSHB														37	4	1	B → M <sub>SP</sub> , SP - 1 → SP	
Pull Data	PULA														32	4	1	SP + 1 → SP, M <sub>SP</sub> → A	
	PULB														33	4	1	SP + 1 → SP, M <sub>SP</sub> → B	
Decrement Stack Pntr	DES														34	4	1	SP - 1 → SP	
Increment Stack Pntr	INS														31	4	1	SP + 1 → SP	
Load stack Pntr	LDS	8E	3	3	9E	4	2	AE	6	2	BE	5	3					M → SP <sub>H</sub> , (M + 1) → SP <sub>L</sub>	
Store Stack Pntr	STS				9F	5	2	AF	7	2	BF	6	3					SP <sub>H</sub> → M, SP <sub>L</sub> → (M + 1)	
Indx Reg → Stack Pntr	TXS														35	4	1	X - 1 → SP	
Stack Pntr → Indx Reg	TSX														30	4	1	SP + 1 → X	

Following are some examples of how the stack can be used. First consider a trivial example. Using only stack operations, swap the contents of accumulators A and B. Assuming the stack pointer has already been set up, the program segment might look like this:

```

PSHA
PSHB
PULA
PULB

```

Assume that accumulator A initially contains  $AA_{16}$  and that accumulator B contains  $BB_{16}$ . The first instruction pushes  $AA_{16}$  onto the stack. Next  $BB_{16}$  is pushed onto the stack. The third instruction pulls  $BB_{16}$  from the top of the stack and places it in accumulator A. Finally, the last instruction pulls  $AA_{16}$  from the stack and places it in accumulator B. As you can see, the contents of the two accumulators are reversed. The following routine accomplishes the same thing with one less instruction:

```

PSHA
TBA
PULB

```

Figure 6-7.  
Stack and stack pointer instructions.

Now look at a more complex example. Assume that you wish to transfer  $16_{10}$  bytes of data from one place in memory to another. As you saw in the previous unit, this type of problem is a good candidate for indexing. However, indexing alone becomes cumbersome if the two lists are over  $FF_{16}$  memory locations apart. The reason for this is that the offset address can only extend  $FF_{16}$  locations above the address in the index register.

In this example, assume you wish to move the data in memory locations  $0010_{16}$  through  $001F_{16}$  to locations  $01F0_{16}$  to  $01FF_{16}$ . While this could be accomplished using indexing alone, the program becomes unnecessarily complicated. Two separate indexes must be maintained; one for loading data from  $0010_{16}$  through  $001F_{16}$ , the other for storing data in  $01F0_{16}$  through  $01FF_{16}$ . A simpler approach is to use indexing for one operation and the stack capability for the other operation. That is, we could load data from the lower list using indexing and store it in the upper list using the stack capability.

A program that does this is shown in Figure 6-8. The first instruction loads the stack pointer with address  $01FF_{16}$ . This is the address of the last entry in the new list that will be formed. Recall that the new list is to be written in locations  $01F0_{16}$  through  $01FF_{16}$ . Once location  $01FF_{16}$  is established as the top of the stack, we can enter data into the new list simply by pushing data onto the stack. Because the stack pointer is decremented with each push operation, we must push the last entry in the list onto the stack first.

Figure 6-8.  
Moving a list of data using both  
indexing and stack operations.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0020	8E	LDS#	Load the stack pointer immediately with the
0021	01	01	address of the last entry in the
0022	FF	FF	new list.
0023	CE	LDX#	Load the index register immediately with the
0024	00	00	address of the last entry in the
0025	1F	1F	original list.
0026	A6	LDAA, X	Load accumulator A indexed from
0027	00	00	the original list.
0028	36	PSHA	Push the contents of accumulator A into the new list.
0029	09	DEX	Decrement the index register.
002A	8C	CPX#	Compare the contents of the index register
002B	00	00	with one less than the address of the
002C	0F	0F	first entry in the original list.
002D	26	BNE	If no match occurs, branch back
002E	F7	F7	this far.
002F	3E	WAI	Otherwise, wait.

The second instruction loads the index register with the address of the last entry in the original list. This is necessary for the reason pointed out above.

Next, the A accumulator is loaded using indexed addressing. Since the offset address is  $00_{16}$ , the accumulator is loaded with the contents of  $001F_{16}$ . That is, the last entry in the original list is loaded into accumulator A.

The PSHA instruction then pushes the contents of accumulator A onto the stack. Thus, the last entry in the original list is transferred to location  $01FF_{16}$ . In the process, the stack pointer is automatically decremented to  $01FE_{16}$ .

The index register is decremented to  $001E_{16}$  by the next instruction. Then, the CPX instruction compares the index register with  $000F_{16}$  to see if all entries in the list have been moved. If no match occurs, the MPU branches back and picks up the next entry in the list. The loop is repeated over and over again until the entire list has been moved to its new location.

Other uses of the stack will be revealed later. However, even if the stack did nothing more than has already been explained, it would be a very useful capability to have. But as you will see, the MPU uses the stack in several other ways that makes this capability even more important.

## Self-Test Review

1. What is a stack?
2. What is a cascade stack?
3. What is a memory stack?
4. Which type of stack does the 6800 MPU use?
5. What is the name of the instruction that stores data in the stack?
6. What type of instruction is used to retrieve data from the stack?
7. What is the purpose of the stack pointer?
8. The PUSH instruction transfers data from one of the accumulators to \_\_\_\_\_.
9. The PULB instruction transfers data from the top of the stack to \_\_\_\_\_.
10. Refer to Figure 6-8. How can we change this program so that the new list is placed in addresses  $0220_{16}$  through  $022F$ ?

## Answers

1. A stack is a group of registers or a section of memory that is used as a last-in, first-out memory.
2. A cascade stack is a group of hardware registers (usually 16 or less) that is used as a last-in, first-out memory.
3. A memory stack uses a section of RAM as a last-in, first-out memory.
4. A memory stack
5. PUSH
6. PULL
7. The stack pointer indicates the address of the top of the stack.
8. The top of the stack.
9. Accumulator B.
10. By changing the first instruction to: LDS# 022F<sub>16</sub>.

## SUBROUTINES

A subroutine is a group of instructions that performs some limited but frequently required task. A given subroutine may be used many times during the execution of the main program. In many cases, the easiest way to write a program is to break the overall job down into many simple operations, each of which can be performed by a subroutine.

Because subroutines are used so frequently, most microprocessors have special capabilities that allow them to handle subroutines efficiently. In this section, these capabilities will be examined. The discussion will start with the instructions associated with subroutines.

The 6800 MPU has three instructions that are used to handle subroutines. They are:

Jump to Subroutine (JSR)  
Branch to Subroutine (BSR)  
Return from Subroutine (RTS)

Each of these will be discussed in this section. One other instruction that has not yet been mentioned will also be discussed. It is the Jump (JMP) instruction. While not used exclusively with subroutines, the JMP instruction makes an excellent introduction to the Jump to Subroutine (JSR) instruction. Therefore, the Jump (JMP) instruction will be discussed first.

### Jump (JMP) Instruction

This instruction allows the MPU to jump from one point in a program to another. In this respect, it is somewhat like the Branch Always (BRA) instruction that was discussed earlier. The difference is the method of addressing used. Recall that the BRA instruction used relative addressing. This has the advantage that only a 2-byte instruction is required. Its disadvantage is that the branch must be within the range of -128 bytes to +127 bytes of the program count.

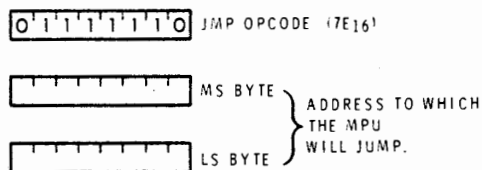


Figure 6-9.  
Format of the JMP instruction  
using extended addressing.

The JMP instruction can use either the indexed or the extended addressing mode. It does not use relative addressing. When using extended addressing, the format of the JMP instruction is as shown in Figure 6-9. Three bytes are required; the opcode followed by the 2-byte address to which the MPU is to jump. Since a 16-bit address is given, the jump may be to any point in the  $65,536_{10}$  byte memory range. This address is loaded into the program counter so that the next opcode is fetched from that address. The previous contents of the program counter are lost. Thus, the MPU starts executing instructions from a new point in memory.

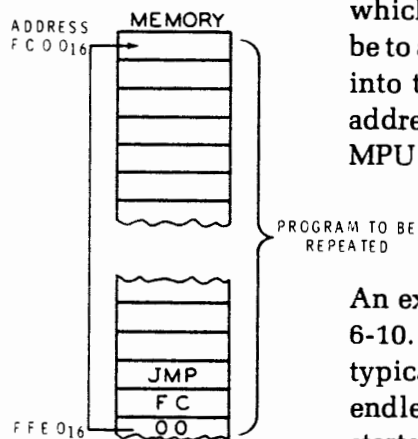


Figure 6-10.  
Using the JMP instruction to  
repeat a program.

An example of how the JMP instruction can be used is shown in Figure 6-10. Here, a long program is to be repeated over and over again. This is typical of applications such as controllers that repeat the same operations endlessly. The program is contained in the upper 1k bytes of memory. It starts at location  $FC00_{16}$  and ends at  $FFE0_{16}$ . Notice that the last instruction is JMP  $FC00_{16}$ . This sends the program back to its beginning so that the loop is repeated endlessly.

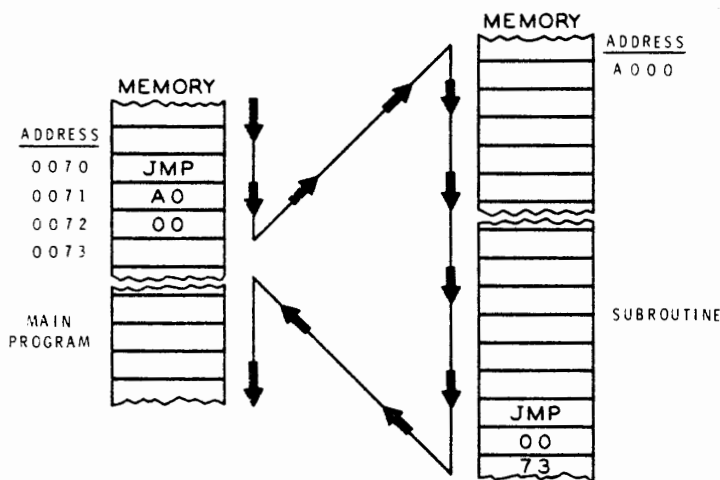


Figure 6-11.  
Using the JMP instruction to  
call a subroutine.

Another possible use of the JMP instruction is shown in Figure 6-11. Here, the main program is in the lower memory locations shown on the left. The main program requires a subroutine that is up at address A000 (shown on the right). The JMP instruction at address 0070 sends the MPU off to the subroutine as shown. The last instruction in the subroutine is another JMP instruction that sends the MPU back to the main program.

Jumping to a subroutine is often referred to as *calling* a subroutine. While we can call a subroutine using the JMP instruction, this approach has a distinct problem. What happens if the main program wants to call the same subroutine more than once? That is, suppose a situation like that shown in Figure 6-12 is required. Here, the main program (on the left) wishes to call the subroutine (on the right) at two separate points. Jumping to the subroutine is no problem. We can do that as many times as we like, using the instruction JMP A000. The problem is: how do we get back from the subroutine to the main program? The first time through the subroutine, the MPU should return to address 0073. The second time through, the MPU should return to address 0093.

A programmer could get around this problem by changing the last instruction in the subroutine before each call or by constructing a table of return addresses, etc. However, most microprocessors have some instructions that solve this problem for us. The following section will discuss the 6800 MPU's solution to this problem.

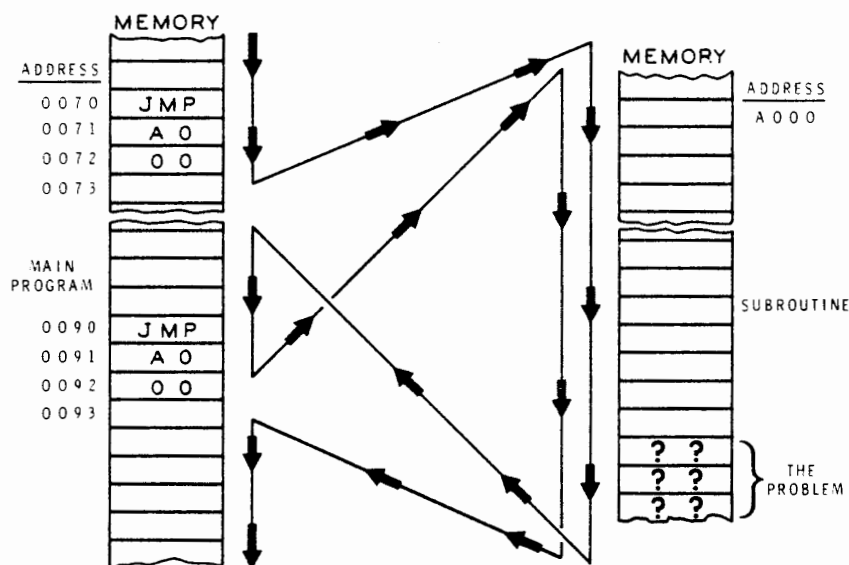


Figure 6-12.  
The JMP instruction cannot  
handle situations like this one.





When the first JSR instruction is executed, the subroutine address  $A000_{16}$  is placed in the program counter. However, just prior to this, the program counter was incremented to the address of the next instruction in sequence. That is, the program counter was advanced to  $0073_{16}$  while the contents of address  $0072_{16}$  were being retrieved. This count ( $0073_{16}$ ) is automatically pushed onto the stack. By saving the old program count, the MPU can tell where to return after the subroutine is finished. As soon as the old program count is tucked away safely in the stack, the subroutine address  $A000_{16}$  is placed in the program counter. Thus, the MPU fetches the next instruction from address  $A000_{16}$ .

Notice that the last instruction in the subroutine is an RTS instruction. When the MPU encounters this single-byte instruction, it will jump back to the point where it left off in the main program. It does this by pulling the old program count ( $0073_{16}$ ) from the stack and placing it in the program counter. Consequently, the next instruction will be fetched from address  $0073_{16}$ . As you can see, this returns the MPU to the correct point in the main program.

Notice that the programmer does not specify a return address at the end of the subroutine. The return address is automatically pulled from the stack. This allows us to call the subroutine repeatedly from several different points in the main program.

Figure 6-13 shows that the subroutine is called again by the JSR  $A000$  instruction in location  $0090_{16}$ . As this instruction and address are decoded, the program count is incremented to  $0093_{16}$ . This program count is pushed onto the stack. Then  $A000_{16}$  is placed in the program counter. Thus, the MPU jumps off to the subroutine. When the subroutine is finished, the RTS instruction causes the old program count to be pulled from the stack into the program counter. This causes the MPU to jump back to address  $0093_{16}$  which contains the next instruction in the main program.

## Nested Subroutines

Figure 6-14 shows a situation in which the main program calls subroutine A. In turn, subroutine A calls subroutine B. In this situation, subroutine B is called a *nested* subroutine. That is, a nested subroutine is a program segment that is called by another subroutine. If control is to be eventually returned to the main program, two program counts must be saved. Figure 6-14 shows how the two program counts are saved in the stack.

At the start of the main program, the stack pointer is loaded with the address of the area in memory that has been set aside to act as the stack. If no stack instructions have been executed when the main program arrives at the first JSR instruction, the stack pointer will still be pointing to where it was originally set. The contents of the stack are of no interest until this point.

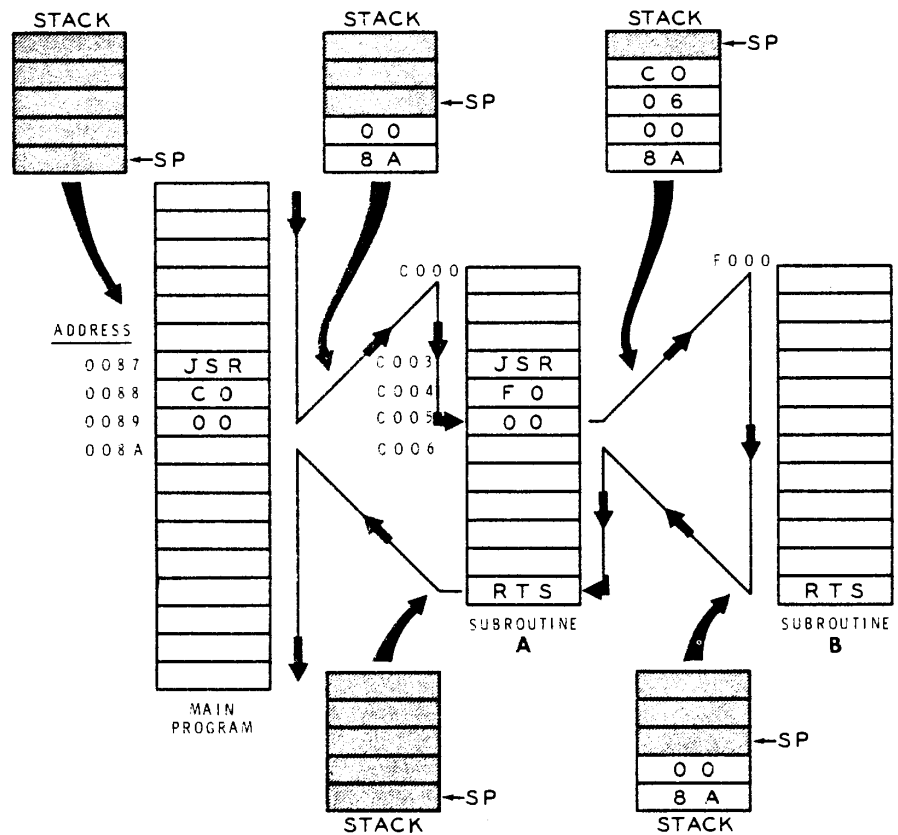


Figure 6-14.  
Handling nested subroutines.

When the main program reaches the JSR instruction, the program count is advanced to the address of the next instruction in sequence ( $008A_{16}$ ). When the JSR instruction is executed, this address ( $008A_{16}$ ) is pushed onto the stack as shown. The low order byte goes in first, followed by the high order byte. In the process, the stack pointer is decremented twice. Finally, the new address ( $C000_{16}$ ) is placed in the program counter. This causes the MPU to jump off to subroutine A which starts at  $C000_{16}$ .

Notice that halfway through subroutine A, subroutine B is called. Consequently, the return address in subroutine A ( $C006_{16}$ ) must be saved. That is, when the program reaches the JSR instruction in subroutine A, the return address ( $C006_{16}$ ) is pushed onto the stack as shown. Notice that there are now two return addresses in the stack. The starting address of subroutine B ( $F000_{16}$ ) is then placed in the program counter and the MPU jumps off to this subroutine.

Subroutine B has no nested subroutines of its own, so the program flow is through the subroutine as shown. The last instruction in subroutine B is the RTS instruction. At this point, the MPU pulls the return address ( $C006_{16}$ ) from the top of the stack and places it in the program counter. This causes the MPU to jump back to the instruction at address  $C006_{16}$  in subroutine A.

The remainder of subroutine A is then executed down to the RTS instruction. This instruction causes the MPU to pull the next address ( $008A_{16}$ ) from the stack and place it in the program counter. Notice that this sends the MPU back to the main program.

For simplicity, a single level of subroutine nesting is shown in this example. However, in practice, many levels of nesting may be used. For example, subroutine B could call subroutine C; etc. Any level of nesting can be used as long as enough memory is set aside for the stack. Remember, each return address requires two bytes in the stack.

## Branch to Subroutine (BSR) Instruction

Quite often, the subroutine we wish to call is within the  $-128_{10}$  to  $+127_{10}$  byte range of the relative address. When it is, we can save one byte by using the Branch to Subroutine (BSR) instruction. The execution of BSR is identical to that of JSR except that relative addressing is used. The old program count is saved in the stack before the branch occurs. Thus, the RTS instruction at the end of the subroutine will cause the old program count to be restored.

## Summary of Subroutine Instructions

Figure 6-15 shows the four instructions discussed in this section. Notice that the BSR instruction uses relative addressing. The JMP and JSR instructions can use either indexed or extended addressing. The RTS instruction uses inherent addressing since its address is pulled from the top of the stack.

Find these instructions on your Instruction Set Summary card. The operations performed by these instructions are illustrated under "Special Operations" on the back of the card. Also, Appendix A of this course gives a concise description of the operations performed by each of these instructions.

JUMP AND BRANCH OPERATIONS		MNEMONIC	RELATIVE			INDEX			EXTND			INHER		
			OP	~	#	OP	~	#	OP	~	#	OP	~	#
Branch To Subroutine	BSR	8D	8	2										
Jump	JMP				6E	4	2	7E	3	3				
Jump To Subroutine	JSR				AD	8	2	8D	9	3				
Return From Subroutine	RTS											39	5	1

Figure 6-15.  
Subroutine and jump instructions.

## Self-Test Review

11. What is a subroutine?
12. What addressing modes can the JMP instruction use?
13. How does the JMP instruction differ from the BRA instruction?
14. How does the execution of the JSR instruction differ from that of the JMP instruction?
15. Why is the program count saved when the JSR or BSR instructions are executed?
16. Where is the program count saved?
17. How is the stack pointer affected by the JSR instruction?
18. Generally, the last instruction in the subroutine will be a \_\_\_\_\_ instruction.
19. What is a nested subroutine?
20. How is the stack pointer affected by the RTS instruction?

## Answers

11. A subroutine is a group of instructions that performs some specific, limited task that is used more than once by the main program.
12. Indexed and extended.
13. Since the BRA instruction uses relative addressing, it can branch only in a  $-128_{10}$  to  $+127_{10}$  byte range. The JMP instruction uses indexed or extended addressing. Therefore, it can jump to any point in memory.
14. When the JSR instruction is executed, the program count is saved in the stack.
15. The program count is saved so that when the subroutine is finished, the MPU can return to the point it left off.
16. The program count is pushed into the top two locations of the stack.
17. The stack pointer is automatically decremented twice as the program count is pushed onto the stack.
18. Return from Subroutine (RTS).
19. When subroutine A calls subroutine B, subroutine B is said to be nested.
20. The stack pointer is automatically incremented twice as the old program count is pulled from the stack.

## INPUT — OUTPUT (I/O) OPERATIONS

A full explanation of input-output (I/O) operations will be given in the next units, but a brief introduction to I/O is necessary at this point. In this section, you will learn what is involved in sending data to — or taking data from — the MPU.

To be useful, a microprocessor system must accept data from the outside world, process it in some way, and present results to the outside world. The input device may be nothing more than a group of switches while the output device can be as simple as a bank of indicator lamps. On the other hand, a single microprocessor might handle several teletypewriters, printers, papertape machines, etc. The point is that the I/O requirements can vary greatly from one application to the next. This section will be concerned with the simplest form of I/O operations.

In the short history of microprocessors, two distinctly different methods have been developed for handling I/O operations. In some microprocessors, I/O operations are handled by I/O instructions. These microprocessors generally have one input instruction and one output instruction. When the input instruction is executed, a byte is transferred from the selected I/O device to a register (usually one of the accumulators) in the MPU. The I/O device is selected by sending out a device selection byte on the address bus. By using an 8-bit byte for device selection, the MPU can specify up to  $256_{10}$  different I/O devices. Of course, no microprocessor system uses that many devices, but the capability is there. The output instruction causes a data transfer from the accumulator to the selected I/O device. While this method of handling I/O operations is used in many microprocessors, the 6800 MPU uses a different technique.

The other method for handling I/O operations is to treat all I/O transfers as memory transfers. This is the method used by the 6800 MPU and many other microprocessors. In fact, even those microprocessors that have I/O instructions can ignore those instructions and handle I/O operations as memory transfers.

The 6800 MPU has no I/O instructions. An I/O device is assigned an address and is treated as a memory location. For example, assume that an input keyboard has been assigned an address of  $8000_{16}$ . We can input data into accumulator A by using the instruction:

LDAA  $8000_{16}$

By the same token, an output display may have been assigned the address  $9000_{16}$ . In this case, we can output from accumulator B by using the instruction:

STAB  $9000_{16}$ .

As you can see, the I/O device is treated as a memory location. The system block diagram shown in Figure 6-16 shows how an I/O device is connected to the microcomputer. Notice that both the data bus and the address bus connect to the I/O interface. As you will see in the next unit, the interface can consist of an address decoder, an output or input latch, and buffers or drivers.

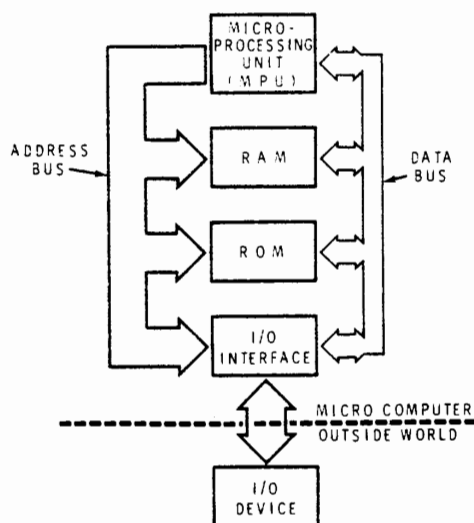


Figure 6-16.  
Adding I/O to the microcomputer.

The address decoder monitors the address bus and enables the interface circuitry whenever the proper address is detected. This prevents the I/O interface from interfering when data is being transferred between memory and the MPU.

The I/O interface will generally have an output latch if it is to be used for an output operation. The reason for this is that the data from the MPU will appear on the data lines for only an instant (usually less than one microsecond). By storing the output data in a latch, the I/O device is given a much longer period of time to examine and respond to the data.

Buffers or drivers are also included in the I/O interface. As you will see later, these are frequently necessary when several different circuits are sharing the same bus.



## Output Operations

Figure 6-17 shows a simplified output circuit. Here, the output device is a bank of eight light emitting diodes (LEDs). Enough detail is shown to illustrate how an output operation can be performed. The address decoder monitors the address bus, looking for the address  $9000_{16}$ . It also monitors some of the control lines that connect to the MPU. One of those lines is called a read-write line. It goes to its low state when a write (output) operation is initiated by the MPU. The other control lines will be discussed in the next unit.

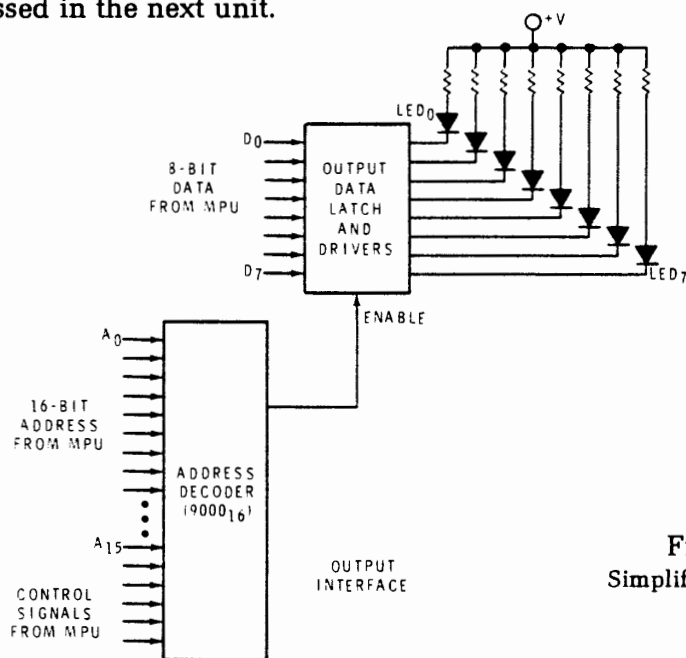


Figure 6-17.  
Simplified output circuit.

Notice that the output of the address decoder is used to enable the output data latch and drivers. When these are enabled, the byte on the data lines is stored in the latch. The data bits stored in the latch cause the appropriate LEDs to light up. By outputting appropriate bit patterns, the MPU can cause different binary numbers to be displayed.

Notice that the address decoder (and therefore the display) is given the address  $9000_{16}$ . We can output data to the display in several different ways. For example, we can load the appropriate pattern to be displayed into accumulator A. Then by executing a "store accumulator A" extended instruction, we can transfer the contents of the accumulator to the display. The instruction would be: **STAA 9000<sub>16</sub>**. Or, we could output data from accumulator B by using the instruction: **STAB 9000<sub>16</sub>**.

In either case, the address  $9000_{16}$  goes out on the address bus for a brief interval of time. The address decoder recognizes this address. At the same time, the control lines indicate that an output operation is called for. In particular, the read-write line goes low. This causes the address decoder to enable the output data latch for an instant. Simultaneously, the 8-bit data byte appears on the data bus. The output latch stores the data byte. The data appears at the input of the latch for less than a microsecond (typically). However, once the data is stored, it appears at the output of the latch until new data is written in. Thus, the output data will be displayed until the next byte of data is outputted by the MPU.

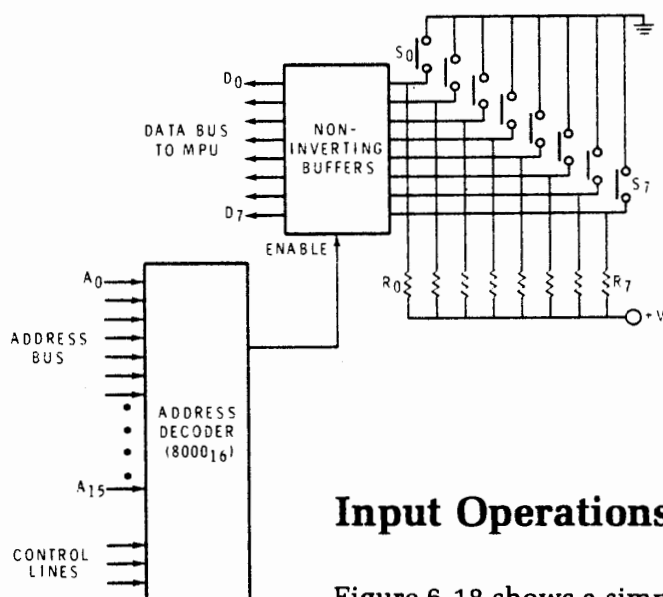


Figure 6-18.  
Simplified input circuit.

## Input Operations

Figure 6-18 shows a simplified input circuit. Here, the input device is a bank of eight switches. When a switch is open, its respective input line to the buffer is held high by the pull-up resistors. However, when a switch is closed, its respective input line is pulled low because the switch connects it to ground.

In this simple circuit, no latch is required between the switches and the data bus. However, a buffer is used so that the switch bank can be effectively disconnected from the data bus when the switches are not being addressed.

As with the output circuit, an address decoder monitors the address and control lines. Notice that the assigned address is  $8000_{16}$ . To input data from the switch bank to accumulator A, we use the instruction: LDAA  $8000_{16}$ . Or, we could input the data to accumulator B by using the instruction: LDAB  $8000_{16}$ .

In either case, the address  $8000_{16}$  is placed on the address line. The address decoder recognizes this address and enables the buffer. For a brief interval (typically less than one microsecond), the lines of the data bus assume the same state as the lines on the right side of the buffer. If no switch is depressed, all data lines will be high and all 1's ( $FF_{16}$ ) will be loaded into the accumulator. However, if one of the switches ( $S_0$ , for example) is depressed, its respective data line ( $D_0$ ) will be low. In this case, the number read into the accumulator will be  $FE_{16}$ . By examining the byte that is read in, the MPU can determine which switch is depressed.

## Input — Output Programming

You now know enough about simple input/output circuits to perform some I/O operations. Refer to Figures 6-17 and 6-18. For the first example, assume that you would like one of the LEDs to light when the corresponding switch is pushed. That is,  $LED_0$  should light when  $S_0$  is pushed;  $LED_1$  should light when  $S_1$  is pushed, etc.

If you refer to Figure 6-17, you will see that an LED is caused to light by placing a 0 in the proper bit in the latch. For example, a 0 in bit 0 will cause  $LED_0$  to be forward biased. Thus, the diode will conduct and emit light. Notice that a 1 at bit 0 will not allow the diode to conduct and emit light. Consequently, a 0 turns the LED on and a 1 turns it off.

Refer to Figure 6-18, and you will find that, when one of the switches is closed, its corresponding line goes to 0. If the switch is not closed, its corresponding line is at 1.

If we load data into one of the accumulators from address  $8000_{16}$  and then store the data at address  $9000_{16}$ , the switches will appear to control the LED's. The program could look like this:

```
LDAA  
80  
00  
STAA  
90  
00  
BRA  
F8
```

If  $S_0$ , and only  $S_0$ , is closed when the LDAA 8000 instruction is executed,  $11111110_2$  will be loaded into accumulator A. The next instruction stores this data byte in the output latch. This causes LED<sub>0</sub>, and only LED<sub>0</sub>, to light. The BRA instruction holds the MPU in a tight loop. Try a few examples and verify that each time a switch is closed, the corresponding LED will light. If the switches are set to some 8-bit binary number, the LED's will display that 8-bit number.

Now, suppose we change our mind and decide that the LEDs should display the one's complement of the binary number set on the switches. We do not have to touch the hardware. Instead, we just change the program. The new program might look like this:

```
LDAA  
80  
00  
COMA  
STAA  
90  
00  
BRA  
F7
```

Notice that we have simply inserted the one's complement instruction between the input and output operations.

As another example, suppose we wish to display a number that is four times greater than the number set on the switches. Our program could be changed to this:

```
LDAA  
80  
00  
ASLA  
ASLA  
STAA  
90  
00  
BRA  
F6
```

Once again, no hardware change is needed. We simply insert two ASLA instructions between the input and output operations.

Although these examples are very simple, they illustrate the flexibility of this I/O arrangement. Data is pulled from the input device as if it were being pulled from memory. Once in the MPU, the data byte can be modified in any way we like. The data can then be transferred to the output device as if it were being stored in memory. While the data is in the MPU, it can be modified in any number of ways. The input byte can be shifted left or right. It can be added to — or subtracted from — another number. It can be ANDed or ORed with another byte. The possibilities are endless and yet none of these involve a hardware change. All data manipulations can be accomplished by the program.

## **Program Control of I/O Operations**

In the preceding examples, all I/O transfers are controlled by the program and the program alone. The program is in a tight loop that inputs data from the switches, modifies the data (if required), and outputs the data to the displays.

When this arrangement is used, the MPU never knows if the data at the input has changed. It simply reads in the data a number of times each second. By the same token, the MPU outputs the data over and over again. This system works well for simple I/O operations. However, as the I/O requirements become more sophisticated, this technique becomes cumbersome.

The program must be in a loop if it is to repeatedly check for inputs and refresh the output. As the number of data manipulations increase, the loop becomes longer and the MPU must check the inputs less frequently. When several I/O devices are used, it must check each input and refresh each output repeatedly. If the loop becomes too long, the MPU may miss a momentary switch closure. This may be acceptable in some applications but in many others it may be intolerable. Obviously then, a more sophisticated method of handling I/O operations must be available to the microcomputer.

## Interrupt Control of I/O Operations

A more effective way of handling I/O operations involves a concept called *interrupts*. Interrupts are a means by which an I/O device can notify the MPU that it is ready to send input data or to accept output data. Generally, when an interrupt occurs, the MPU suspends its current operation and takes care of the interrupt. That is, it might read in or write out a byte of data. After it has taken care of the interrupt, the MPU returns to its original task and takes up where it left off.

An analogy may help you to visualize an interrupt operation. Compare the MPU to the president of a corporation who is writing a report. The interrupt can be compared to a telephone call. The president's main task is the report. However, if the telephone rings (an interrupt), she finishes writing the present word or sentence then answers the phone call. After she has attended to the phone call, she returns to the report and takes up where she left off. In this analogy, the ringing of the telephone notifies the president of the interrupt request.

This analogy shows the difficulty of the program controlled I/O technique discussed earlier. If we remove the interrupt request (the ringing of the phone), we are left with an almost comical situation. The president writes a few words of the report. She then picks up the phone to see if anyone is on the other end. If not, she hangs up the phone, writes a few more words, and checks the phone again. Clearly, this technique wastes an important resource — the president's time.

This simple analogy shows the importance of an interrupt capability. Without it, a great deal of the MPU's time can be wasted doing routine operations. The next section will examine the interrupt capabilities of the 6800 MPU.

## Self-Test Review

21. What are the two methods by which microprocessors handle I/O operations?
22. Which method does the 6800 MPU use?
23. Which instruction can be used for transferring data from an I/O device to accumulator A?
24. Which instruction can be used for transferring data from accumulator B to an I/O device.
25. Write a program segment that will: read in data from the switch bank shown in Figure 6-18; double the number; and display the result on the LED bank shown in Figure 6-17.
26. What is meant by program control of an I/O operation?
27. What is meant by interrupt control of an I/O operation?

## Answers

- 21. Some microprocessors have input-output instructions; others treat I/O as memory.
- 22. The 6800 MPU treats I/O as memory.
- 23. LDAA
- 24. STAB
- 25. One solution is:

LDAA  
80  
00  
ASLA  
STAA  
90  
00

- 26. Using this method, the program regularly reads in or writes out data. All I/O operations are controlled by the program.
- 27. Using this method, the I/O device itself signals the MPU that it is ready to transmit or receive data. The I/O operations are controlled largely by the I/O device itself.



## INTERRUPTS

Interrupts were introduced in the previous section in connection with I/O operations. While I/O operations use part of the interrupt capability of the MPU, interrupts are also used in other ways. The 6800 MPU has four different types of interrupts:

Reset  
Non-Maskable Interrupt (NMI)  
Interrupt Request (IRQ)  
Software Interrupt (SWI)

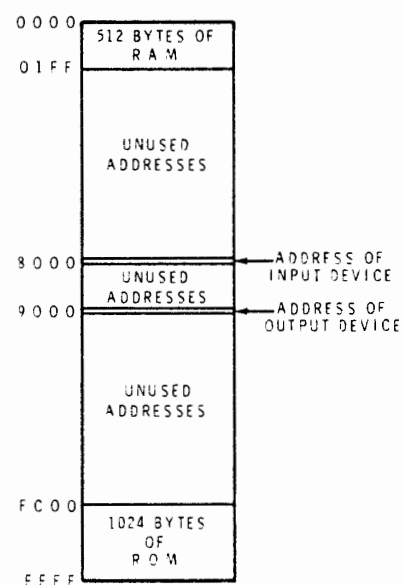
This section will examine each of these interrupts in detail.

### Reset

In a typical application, the microcomputer has a control or monitor program in a read-only-memory (ROM). Also, a random access read-write memory (RAM) is used for holding input data, intermediate answers, output data, etc. As we have seen, the 6800 MPU has the capability of addressing up to  $65,536_{10}$  memory locations. Most microprocessor applications do not require this much memory. In many applications, the control program requires less than ten percent of the possible locations. The RAM probably uses less than two percent. Generally, the monitor program is placed at the high memory addresses. The RAM is usually given the low memory addresses so that the direct addressing mode can be used. The I/O devices are given intermediate addresses. Thus, the memory addresses may be allocated as shown in Figure 6-19.

Notice that the control or monitor program is placed in a ROM at the very top of memory. In this example, a  $1024_{10}$  byte ROM is used. The addresses of the ROM are  $FC00_{16}$  through  $FFFF_{16}$ . A small RAM is placed at the low end of memory. Addresses  $0000_{16}$  through  $01FF_{16}$  are used. Notice that all other addresses are unused except for two. The input device is assigned address  $8000_{16}$ , while the output device is assigned address  $9000_{16}$ .

The monitor program stored in the ROM, controls all the activities of the MPU. At all times, the entire system is being run by this program. In this example, when the microprocessor is initially turned on, it should start executing instructions at address  $FC00_{16}$ . Also, we should be able to restart the program at this address at any time. In order to accomplish this, the 6800 MPU has a built-in reset capability.



**Figure 6-19.**  
Memory allocations in a typical microcomputer system.

The 6800 MPU has a signal line or control pin that is called  $\overline{\text{Reset}}$ . This pin or line is connected to a reset switch of some kind. If this line goes low for a prescribed period of time (to be explained later) and then swings high, the MPU will initiate a reset interrupt sequence. The main purpose of the reset interrupt sequence is to load the address of the first instruction to be executed into the program counter. This would be easy to accomplish if, in every application, the starting address were the same. However, the starting address differs from one application to the next. Therefore, a convenient means is provided to allow the designer to specify any starting address that he likes.

In any 6800 based microprocessor system, the upper eight bytes of ROM are reserved for interrupt vectors. An interrupt vector is simply an address that is loaded into the program counter when an interrupt occurs. Figure 6-20 shows how these eight reserved memory bytes are allocated. Notice that addresses  $\text{FFFE}_{16}$  and  $\text{FFFF}_{16}$  contain the reset vector. That is, these two memory locations contain the address of the first instruction that is to be executed when the microcomputer is initially started. In our example, the first instruction in the monitor program is at address  $\text{FC00}_{16}$ . Consequently, this is our reset vector. Location  $\text{FFFE}_{16}$  must contain the high byte of the address ( $\text{FC}_{16}$ ) and  $\text{FFFF}_{16}$  must contain the low byte of the address ( $\text{00}_{16}$ ).

Remember locations  $\text{FFFE}_{16}$  and  $\text{FFFF}_{16}$  are in the read-only-memory. Therefore, the designer must provide the proper reset vector at the time he is writing the monitor program.

Figure 6-20.  
Interrupt vector assignments.

Address	
F F F 8	Interrupt Request Vector (high order address)
F F F 9	Interrupt Request Vector (low order address)
F F F A	Software Interrupt Vector (high order address)
F F F B	Software Interrupt Vector (low order address)
F F F C	Non-Maskable-Interrupt Vector (high order address)
F F F D	Non-Maskable-Interrupt Vector (low order address)
F F F E	Reset Vector (high order address)
F F F F	Reset Vector (low order address)

Figure 6-21 shows the sequence of events that occurs when the MPU is reset. First, the interrupt (I) mask bit is set. You will recall that the I flag is one of the condition code registers. As you will see later, if this flag is set, it prevents one of the other interrupts from occurring. Thus, the MPU sets the interrupt mask bit so that the reset sequence will not be interrupted by a request for interrupt by one of the I/O devices.

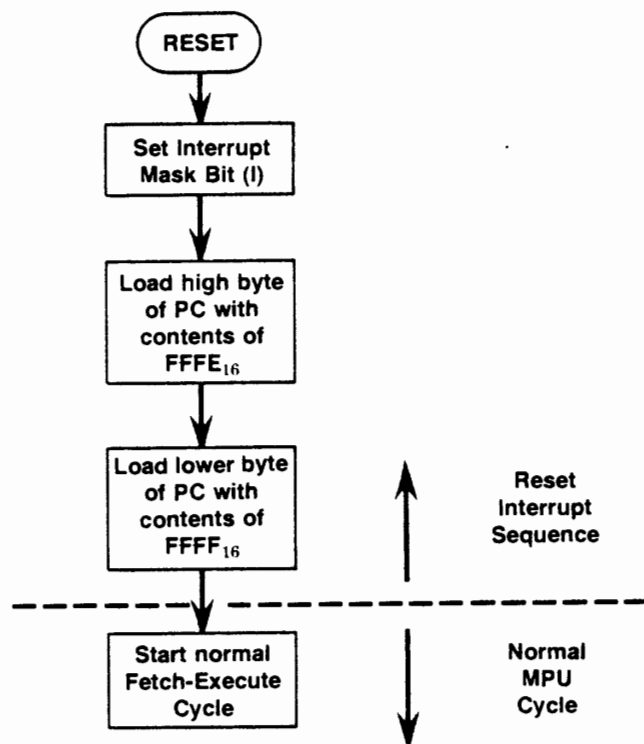


Figure 6-21.  
Reset interrupt sequence.

Second, the contents of location FFFE<sub>16</sub> are loaded into the high byte of the program counter. This is done by sending the address FFFE<sub>16</sub> out on the address bus. The memory location is read out and its contents are placed on the data bus. The MPU picks up this byte and places it in the upper eight bits of the program counter. In our example, the byte in location FFFE<sub>16</sub> is FC<sub>16</sub>.

Next, the contents of location FFFF<sub>16</sub> are loaded into the lower eight bits of the program counter. This is done by setting the address bus to FFFF<sub>16</sub>. Thus, the contents of the highest memory location are placed on the data bus. In our example, this byte is 00<sub>16</sub>. At this point, the program counter contains the address of the first instruction which is FC00<sub>16</sub>.

The reset sequence is then terminated by switching the MPU to its normal fetch-execute machine cycle. Thus, the instruction at address FC00<sub>16</sub> is fetched and executed. From this point on, all MPU activities are controlled by the program.

The microprocessor system will have a reset switch somewhere in the system. This will allow the operator to restart the system if the system locks up or runs away for some reason. In addition, some systems will have an automatic reset feature that will allow the system to reset itself after a power failure. In both cases, the reset capability of the MPU is used.

This reset capability can be considered an interrupt, since the MPU leaves whatever it is doing and jumps off to the start of the monitor program. In most cases, the monitor program would start with a short subroutine that initializes the system. It would do things like set up the stack pointer, initialize displays, etc.

## Non-Maskable Interrupts

The 6800 has two other types of hardware interrupts. One of these interrupts is maskable; the other is not. A maskable interrupt is one that the MPU can ignore under certain conditions. Whereas, a non-maskable interrupt cannot be ignored. To illustrate the difference, recall the corporation president analogy.

The president's report writing can be interrupted by the telephone. However, by telling her secretary to hold all calls, she has effectively masked one source of interruptions. In this analogy it is impractical to mask all interrupts. For example, it could be counterproductive to mask the fire alarm.

Somewhat the same situation can exist in a microprocessor controlled system. Some interrupts can be ignored for a few seconds while the MPU is performing a more important task. This type of interrupt can be masked. Others must not be ignored at all. These cannot be masked. Of course, it is up to the designer to decide which interrupts can be masked and which cannot. The 6800 MPU has provisions for handling both types. How the MPU handles the non-maskable type will be discussed first.

The 6800 MPU has a control line called the non-maskable interrupt (NMI) line. A high-to-low transition on this line forces the MPU to initiate a *non-maskable interrupt sequence*. The purpose of this sequence is to provide an orderly means by which the MPU can jump off to a service routine that will take care of the interrupt.

This becomes somewhat involved because the MPU must be able to go back to its main program after the interrupt service routine is finished. It must be able to pick up exactly where it left off. Furthermore, all registers must hold exactly the same data and addresses that they held when the

interrupt occurred. In other words, when an interrupt occurs, the program count must be saved so that the MPU can later return to this point in the program. Also, the contents of the accumulators, index register, and even the condition code registers must be saved so that the MPU can be restored to the exact condition that existed at the instant the interrupt occurred.

The 6800 MPU accomplishes this by pushing all the pertinent data onto the stack. Then, after the interrupt has been serviced, the MPU returns to its previous status by pulling the data from the stack.

The non-maskable interrupt sequence is shown in Figure 6-22. A non-maskable interrupt is initiated when the  $\overline{\text{NMI}}$  line goes from its high state to its low state. The MPU finishes the execution of the current instruction. However, before another instruction is fetched, the MPU pushes the contents of its registers onto the stack. Recall that the stack pointer always points to the top of the stack. For this example, assume that the stack pointer was set by an earlier instruction to address  $0068_{16}$ .

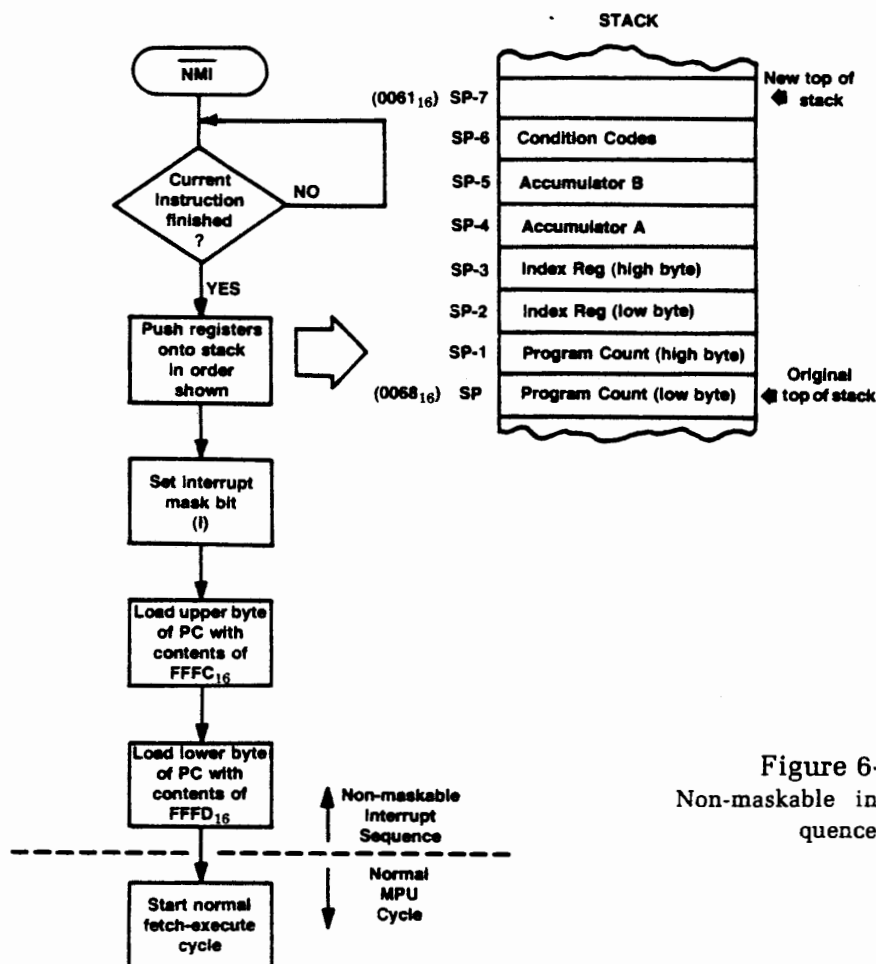


Figure 6-22.  
Non-maskable interrupt sequence.

The MPU pushes the lower eight bits of the program counter into memory location  $0068_{16}$ . Then it decrements the stack pointer so that the upper eight bits of the program counter are pushed into address  $0067_{16}$ . Next, the contents of the index register are pushed into addresses  $0066_{16}$  and  $0065_{16}$ . The contents of accumulators A and B and the condition codes are also pushed in as shown. When all this has been done, the stack pointer will have been decremented seven times to  $0061_{16}$ .

Return to the flow chart and notice that the next step is to set the interrupt mask bit. This allows the MPU to ignore any interrupt requests that occur while the non-maskable interrupt is being serviced.

At this point, the MPU is ready to jump to the interrupt service routine. But, what is the address of this routine? Recall the interrupt vector chart that was shown earlier in Figure 6-20. The non-maskable interrupt vector is at addresses  $FFFC_{16}$  and  $FFFD_{16}$ . Thus, the upper byte of the program counter is loaded from  $FFFC_{16}$  while the lower byte is loaded from  $FFFD_{16}$ . This directs the MPU to the first instruction in the non-maskable interrupt service routine. From this point on, the MPU returns to its normal fetch-execute cycle until the service routine is finished.

The sequence of events shown in Figure 6-22 happen automatically when a non-maskable interrupt sequence is initiated. The  $\overline{NMI}$  line gives external hardware a method of forcing a jump-to-subroutine to occur. In this case, the subroutine is a short program that performs some action to take care of the interrupt.

## Return From Interrupt (RTI) Instruction

The non-maskable interrupt is used when some situation exists that cannot be ignored. You can probably visualize applications that would require such a capability. For example, assume that a microprocessor is being used in a numerically controlled drill press. The non-maskable interrupt could be used in conjunction with limit switches to prevent drilling holes in the work surface. Or, it could be used to shut down the machine if someone's hand got too close.

The purpose of the service routine is to direct the operation of the computer to take care of the interrupt. Typically, it would first determine which external device initiated the interrupt. Then it would determine the nature of the interrupt. Finally, it would take whatever action was necessary to take care of the interrupt. In many cases, the interrupt is of a routine nature and can be easily serviced. In these situations, the MPU

should return to the main program and take up where it left off. There is an instruction that allows the MPU to do this. It is called the "Return-From-Interrupt" (RTI) instruction. Look on your Instruction Set Summary card, and you will see that this is a one-byte instruction whose opcode is  $3B_{16}$ .

Figure 6-23 shows how the RTI instruction is used. The main program is shown on the left, while the interrupt service routine is shown on the right. Assume that the interrupt signal occurs while the LDAB# instruction is being executed. The MPU finishes that one instruction and pushes all pertinent data onto the stack. It then jumps to an address determined by the  $\overline{NMI}$  vector in address FFFC and FFFD. The contents of these two locations determine the starting address of the  $\overline{NMI}$  service routine. Notice that the last instruction in the service routine is the return-from-interrupt instruction. This instruction returns program control to the point in the main program that the MPU left when the interrupt occurred.

This can be done because the previous status of the MPU was preserved in the stack. The RTI instruction causes the accumulators, the index registers, the condition code register, and the program counter to be loaded from the stack. Thus, the same information that went into the stack when the interrupt occurred comes out of the stack when the RTI instruction is executed. This allows the MPU to return to the main program and take up where it left off.

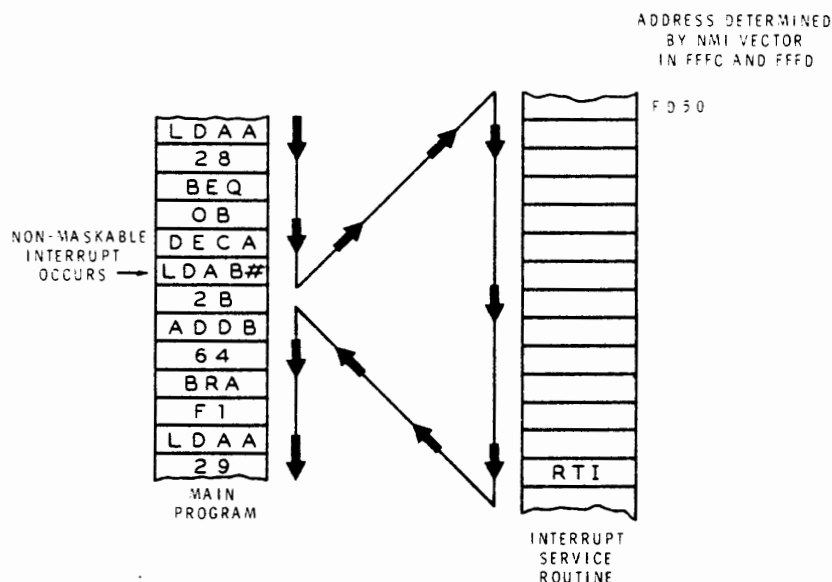


Figure 6-23.

The RTI instruction returns control to the main program after the interrupt has been serviced.

## Interrupt Request (IRQ)

The interrupt request is very similar to the non-maskable interrupt. The main difference between the two is that the interrupt request is maskable.

The 6800 MPU has a control line called the interrupt request ( $\overline{\text{IRQ}}$ ) line. When this line is low, an interrupt sequence is requested. However, the MPU may or may not initiate the interrupt sequence depending on the state of the interrupt mask (I) bit in the condition code register. If the I bit is set, the MPU ignores the interrupt request. If the I bit is not set, the MPU initiates the interrupt sequence. This procedure is very similar to the  $\overline{\text{NMI}}$  procedure discussed earlier. Figure 6-24 shows the interrupt procedure.

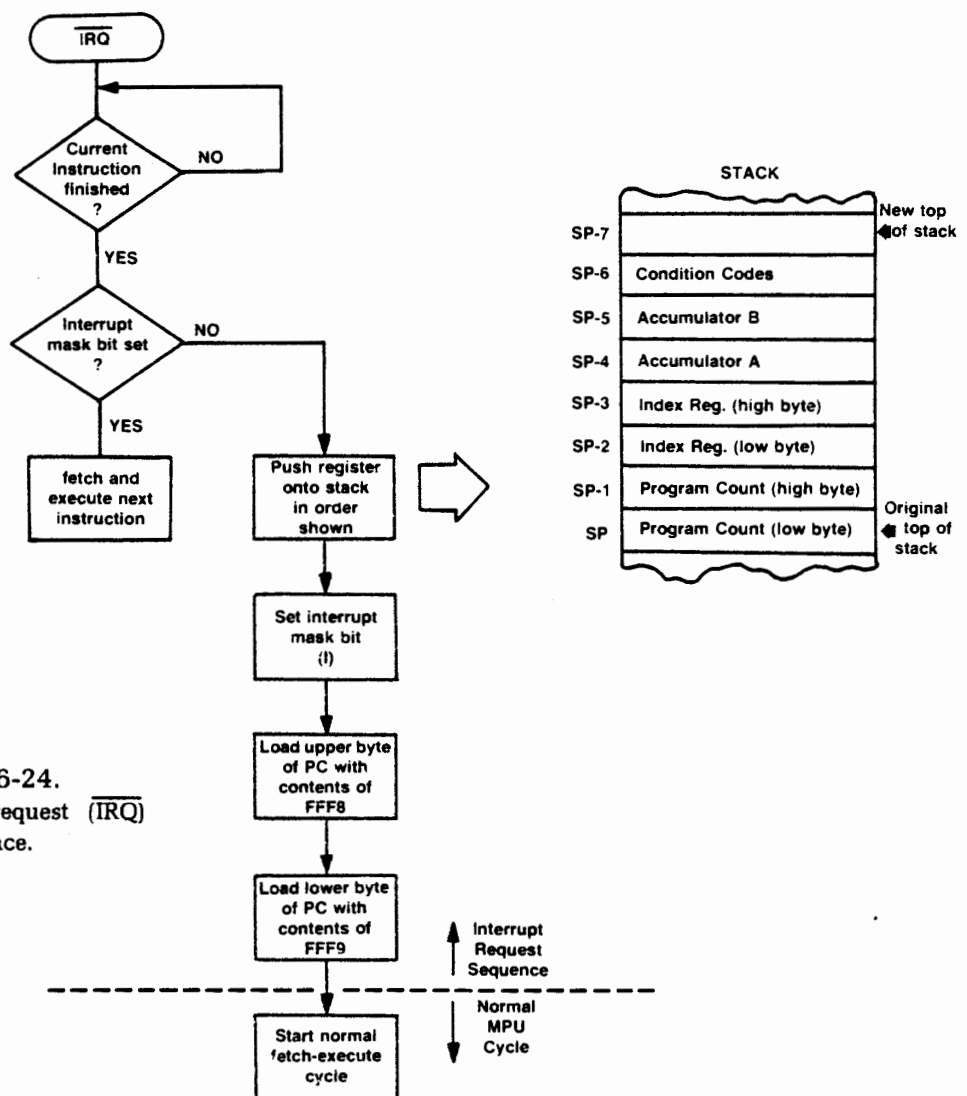


Figure 6-24.  
The interrupt request ( $\overline{\text{IRQ}}$ )  
sequence.



When the  $\overline{\text{IRQ}}$  line is low, the MPU finishes the current instruction. It then checks the interrupt mask bit. If I is set to 1, the MPU ignores the interrupt request and executes the next instruction in sequence. However, if I=0, the MPU pushes the contents of the various registers onto the stack in the order shown.

Next, the interrupt mask bit is set to 1. This prevents the MPU from honoring other interrupt requests until the present interrupt has been serviced.

The address of the  $\overline{\text{IRQ}}$  service routine is at addresses  $\text{FFF8}_{16}$  and  $\text{FFF9}_{16}$ . The program counter is loaded from these addresses. Thus, the next instruction to be executed will be the first instruction in the interrupt request service routine.

Once in the service routine, the MPU goes into its normal fetch-execute cycle. When the interrupt has been serviced, control can be returned to the main program by an RTI instruction.

## Interrupt Mask Instructions

The 6800 MPU has two instructions that allow software control of the interrupt mask bit. You have seen that the I bit in the condition code register is set any time an interrupt sequence is initiated. This prevents an  $\overline{\text{IRQ}}$  from being honored while a previous  $\overline{\text{IRQ}}$  or  $\overline{\text{NMI}}$  is being serviced. This is an example of setting the interrupt flag with hardware.

In many cases, it is necessary to set the interrupt flag with software. Therefore, the 6800 MPU has an instruction that can do this. It is called the "Set-Interrupt-Mask" (SEI) instruction. If you refer to your Instruction Set Summary card, you will see that this is a one-byte instruction whose opcode is  $0\text{F}_{16}$ . The flag may be set to prevent an interruption on a part of the program that we do not wish to be interrupted. It has the effect of disabling interrupt requests.

Of course, we do not wish to permanently disable the interrupt capability. Therefore, some means must be provided for enabling the interrupt request capability. An instruction called "Clear-Interrupt-Mask" (CLI) is available for this purpose. This is a one-byte instruction whose opcode is  $0\text{E}_{16}$ .

While we can disable or enable the interrupt request line with these instructions, they do not affect the non-maskable interrupt. As the name implies, the  $\overline{\text{NMI}}$  line cannot be disabled by the I flag.

## Software Interrupt (SWI) Instruction

The 6800 MPU has a software equivalent of an interrupt. It is an instruction called the "Software Interrupt" (SWI). When executed, the instruction causes the MPU to perform an interrupt sequence that is very similar to the hardware interrupt sequences already discussed. As shown on your Instruction Set Summary card, this is a one-byte instruction whose opcode is  $3F_{16}$ .

Figure 6-25 shows the sequence of events that occurs when this instruction is executed. First the contents of all the pertinent registers are pushed onto the stack in the order shown. Next, the interrupt mask is set so that interrupt requests cannot interfere. Finally, the software interrupt vector is obtained from addresses  $FFFA_{16}$  and  $FFFB_{16}$ . This vector is loaded into the program counter so that the next instruction will be fetched from this address. As with the other interrupts, the MPU will return to the original program when a return-from-interrupt instruction is encountered.

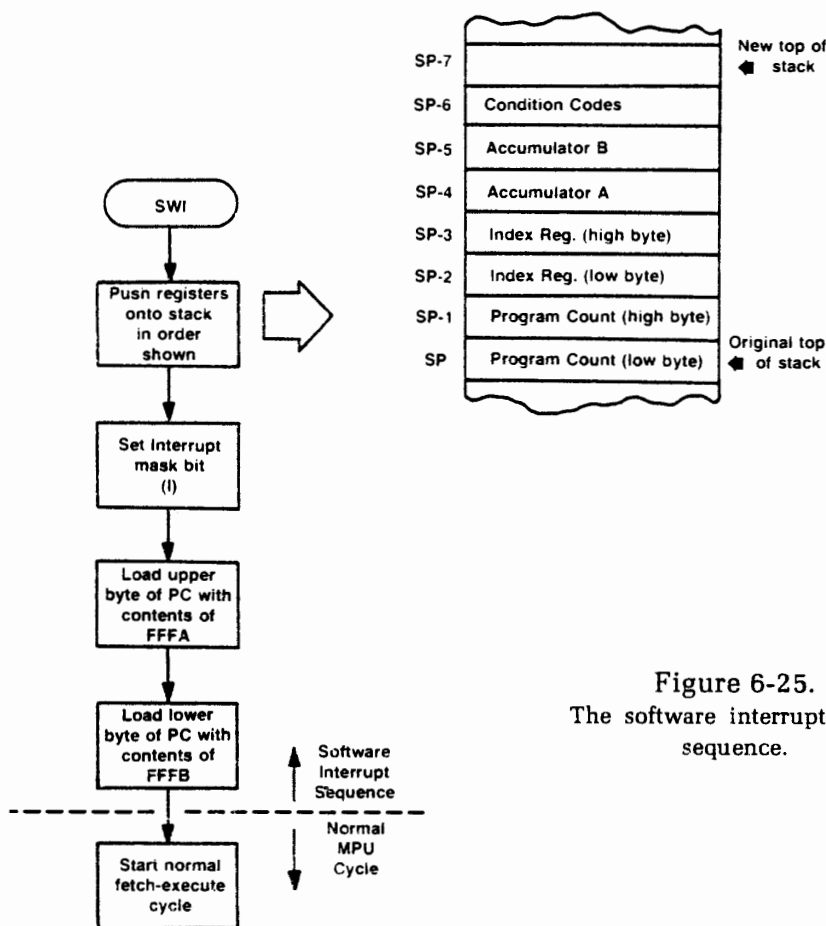


Figure 6-25.  
The software interrupt (SWI)  
sequence.

The software interrupt instruction can be used to simulate hardware interrupts. It is also helpful for inserting pauses in a program. For example, the ET-3400 Microprocessor Trainer uses the software interrupt to perform the single-step function and to implement the breakpoint capability.

## Wait for Interrupt (WAI) Instruction

One of the first instructions introduced in this course was the halt instruction (opcode  $3E_{16}$ ). In the previous unit, you learned that this instruction is actually called a Wait-for-Interrupt (WAI). What exactly does this instruction do? It does cause the MPU to halt, but there is more to it than that.

When the WAI instruction is executed, the program counter is incremented by one. Then the contents of the program counter, index register, accumulators, and condition code register are pushed onto the stack. The order is exactly the same as if an interrupt occurs. The MPU then enters a wait state, doing nothing further until, and unless, an interrupt occurs.

The MPU can be forced back into action either by an interrupt request or by a non-maskable interrupt. The  $\overline{NMI}$  sequence is the same as that described earlier except for one important difference. Remember that the contents of the registers have already been pushed onto the stack. Thus, this part of the  $\overline{NMI}$  sequence is omitted. This allows the MPU to respond faster to the interrupt.

The  $\overline{IRQ}$  sequence is also the same as that described earlier except that the registers are not pushed onto the stack again. As always, the  $\overline{IRQ}$  signal is ignored if the interrupt mask bit is set.

Of course, the reset signal can override the wait state. Thus, there are three ways of escaping the wait state.



## Answers

28. Reset, non-maskable interrupt, interrupt request, and software interrupt.
29. Interrupt request ( $\overline{\text{IRQ}}$ ).
30. To direct the MPU to the first instruction in the monitor or control program.
31.  $\text{FFFE}_{16}$  and  $\text{FFFF}_{16}$ .
32. The address of the interrupt service routine.
33.
  - A. The current instruction is executed.
  - B. The contents of the pertinent registers are pushed onto the stack.
  - C. The interrupt mask bit is set.
  - D. The  $\overline{\text{NMI}}$  vector from addresses  $\text{FFFC}_{16}$  and  $\text{FFFD}_{16}$  is loaded into the program counter.
  - E. The instruction at the address specified by the  $\overline{\text{NMI}}$  vector is fetched and executed.
34. A routine that takes care of the interrupt and then returns control to the main program.
35. The Return-From-Interrupt (RTI) instruction.
36. The stack pointer is incremented seven times as the previous MPU status is pulled from the stack.
37. Reset.
38. Set Interrupt Mask (SEI).
39.
  - A. By a reset signal.
  - B. By a non-maskable interrupt.
  - C. By an interrupt request (if  $\text{I}=0$ ).
40. If the MPU is waiting for an interrupt.

## EXPERIMENTS

Perform Experiments 9 and 10 in the Programming Experiment Section (Unit 9) of this course. After you finish these experiments, return to this unit and complete the unit examination.

## UNIT EXAMINATION

1. If the I bit in the condition code register is set, the MPU will ignore:
  - A. The reset signal.
  - B. The non-maskable interrupt signal.
  - C. The interrupt request signal.
  - D. The software interrupt instruction.
  
2. Which of the following lists contains instructions that do **not** change the contents of the stack pointer?
  - A. PULA, DES, RTI, WAI.
  - B. PSHB, INS, RTS, SWI.
  - C. TXS, BSR, PULB, LDS.
  - D. PSHA, JMP, TSX, STS.
  
3. Which of the following program segments will successfully swap the contents of the two accumulators?
 

A. PSHA	B. PSHB	C. PSHA	D. PSHB
TAB	TAB	TBA	TBA
PULA	PULA	PULA	PULB
  
4. The stack pointer is automatically:
  - A. Decrement before data is pushed onto the stack.
  - B. Increment before data is pushed onto the stack.
  - C. Decrement after data is pushed onto the stack.
  - D. Increment after data is pushed onto the stack.
  
5. One difference between the JMP and JSR instruction is:
  - A. JMP can use either extended or indexed addressing.
  - B. The program count is saved when JSR is executed.
  - C. The JSR will be executed even if the interrupt mask is set.
  - D. JMP is an unconditional jump.
  
6. The last instruction in a subroutine is generally:
  - A. A JMP instruction.
  - B. An RTS instruction.
  - C. An RTI instruction.
  - D. A JSR instruction.

7. In the 6800 MPU, which of the following instructions could be used to transfer data from an I/O device to accumulator A?
- INPA.
  - LDAA.
  - STAA.
  - OUTA.
8. Refer to Figures 6-17 and 6-18. Which of the following program segments will read in data from the switch bank and, if the number is larger than  $2A_{16}$ , display it on the LED's?
- |         |         |         |         |
|---------|---------|---------|---------|
| A. LDAA | B. LDAA | C. LDAA | D. LDAA |
| 80      | 80      | 80      | 90      |
| 00      | 00      | 00      | 00      |
| CMAPA#  | SUBA#   | STAA    | SUBA#   |
| 2A      | 2A      | 90      | 2A      |
| BHI     | BHI     | 00      | BHI     |
| 01      | 01      |         | 01      |
| WAI     | WAI     |         | WAI     |
| STAA    | STAA    |         | STA     |
| 90      | 90      |         | 80      |
| 00      | 00      |         | 00      |
9. Which of the following types of interrupts does **not** cause data to be pushed into the stack?
- Software interrupt.
  - Non-maskable interrupt.
  - Reset interrupt.
  - Interrupt request.
10. Generally, the last instruction in an interrupt service routine will be:
- An RTI instruction.
  - An SWI instruction.
  - An RTS instruction.
  - An NMI instruction.





# Individual Learning Program

## MICROPROCESSORS

*Unit 7*

**INTERFACING — PART 1**

EE-3401

HEATH COMPANY  
BENTON HARBOR, MICHIGAN 49022

Copyright © 1977  
Heath Company  
All Rights Reserved  
Printed in the United States of America

## CONTENTS

Introduction .....	7-3
Unit Objectives .....	7-4
Unit Activity Guide .....	7-5
Interfacing Fundamentals .....	7-6
Interfacing with Random Access Memory .....	7-20
Interfacing with Displays .....	7-36
Interfacing Experiments .....	7-50
Unit Examination .....	7-51
Examination Answers .....	7-53

## INTRODUCTION

As mentioned earlier, there are two things you can do with a micro-processor. You can program it and you can interface it with other circuits. By now you should be able to write simple programs without too much trouble. You will continue to learn new programming techniques as you progress through the course. However, in the remaining portions of the course, the emphasis will shift to interfacing the MPU with other circuits.

Units 7 and 8 will give you enough general information to understand the interfacing experiments providing that you have a general knowledge of digital electronics. That is, these units are written on the assumption that you have training equivalent to that provided in the Heathkit Continuing Education, Individual Learning Program in Digital Electronics.

## UNIT OBJECTIVES

When you have completed this unit, you should be able to:

1. Define 3-state logic and explain the need for it.
2. Explain the purpose of each of the control lines on the 6800 MPU.
3. Explain the timing relationships between the clock signals and the information on the address, data, and  $R/\overline{W}$  lines.
4. Identify several different arrangements used in static RAMS.
5. Draw the logic diagram of a simple address decoder.
6. Explain the operation of a static RAM storage cell.
7. Show four different methods by which an MPU can drive 7-segment displays.

## UNIT ACTIVITY GUIDE

- ☐ Read Section on Interfacing Fundamentals.
- ☐ Complete Self-Test Review Questions 1 through 14.
- ☐ Play Cassette Tape Section "Semiconductor Memories."
- ☐ Read Section on Interfacing with Random Access Memory.
- ☐ Complete Self-Test Review Questions 15 through 28.
- ☐ Read Section on Interfacing with Displays.
- ☐ Complete Self-Test Review Questions 29 through 40.
- ☐ Perform Interfacing Experiments 1 through 4.
- ☐ Complete Unit Examination.
- ☐ Check Examination Answers.

## INTERFACING

### FUNDAMENTALS

Before going into specific interfacing examples, we must first discuss some fundamental concepts that will be used. First, we will discuss the concept of a bus and the need for 3-state logic. Then we will examine the various control and bus lines of the 6800 MPU. Finally, we will consider the various timing relationships involved in the execution of instructions.

### Buses

In computer jargon, a bus is generally defined as a group of conductors over which information is transferred from one place to another. In many cases, the information can originate from any one of several sources and can be transferred to any one of several destinations. Moreover, on some buses, information can be transferred in either of two directions. These are called bi-directional buses. Of course for a given bus, only one transfer of information can occur at a time.

Figure 7-1 shows the data bus arrangement in a typical microcomputer application. Generally in this type of system, all data transfers involve the MPU. Thus, data can be transferred in either direction between RAM and the MPU. However, other data transfers are one way only. Data can be transferred from ROM or the input buffer to the MPU. Also, data can be transferred from the MPU to the output latches.

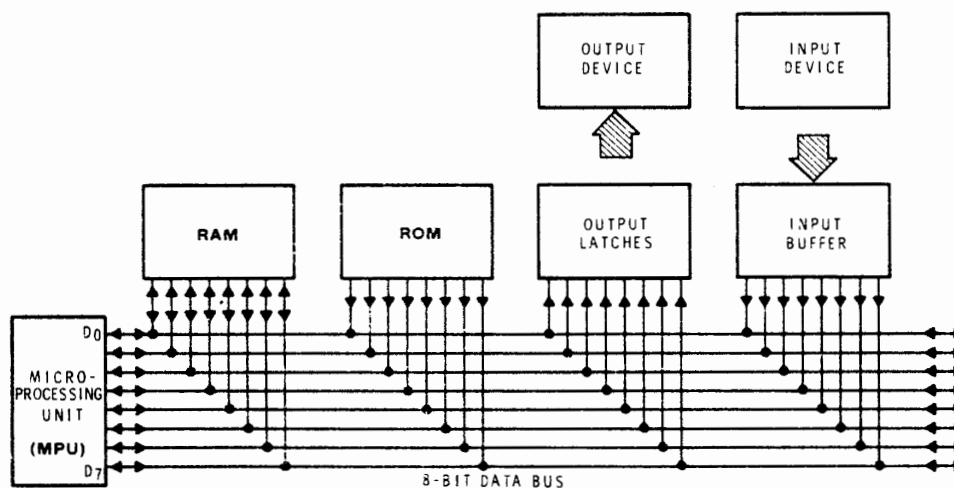


Figure 7-1

Typical Data Bus Arrangement.

In an arrangement like this, two problems arise. First, we must insure that only one data transfer is attempted at any given time. This is done by assigning each destination or source a different address. For example, the RAM, ROM, output latches, and input buffers all have one or more chip enable pins. The proper logic levels on these pins will select or activate the circuit. By assigning each circuit a different address, we insure that only one circuit at a time is enabled.

Figure 7-2 shows the addressing capability added to the block diagram. An address decoder is added for each circuit. The inputs to the address decoders come from the MPU via the address bus. The outputs go to the chip enable lines of the various circuits. Since only one address can appear on the address bus at any given instant, only one of the external circuits will be enabled at a time.

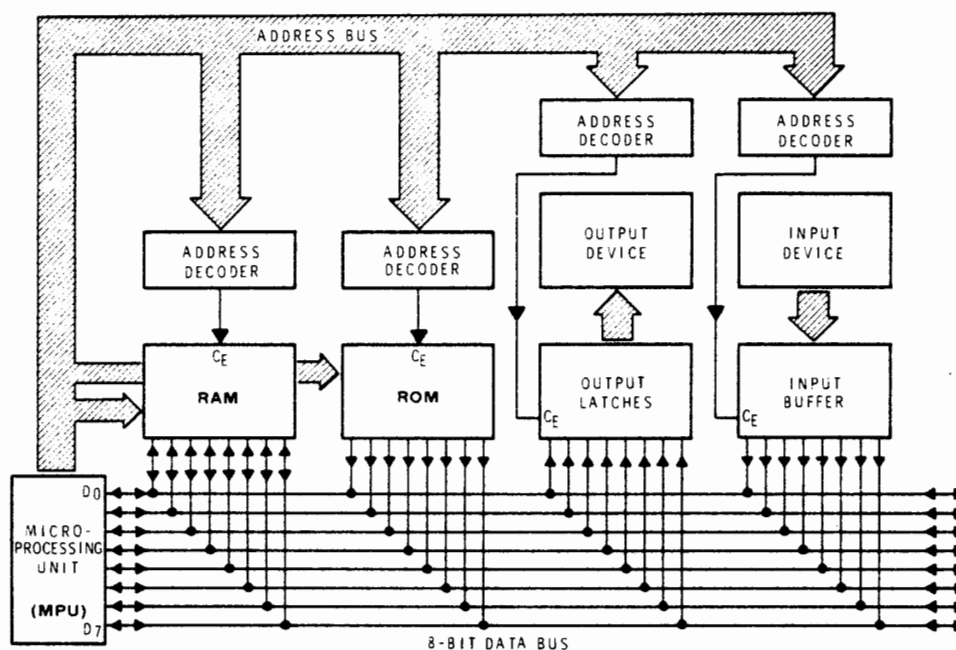


Figure 7-2  
Adding the Address Capability.

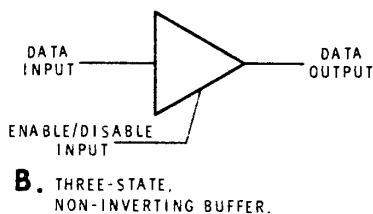
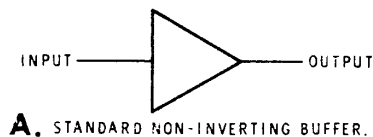
The memories are assigned many addresses since each byte must have its own address. For example, if a  $512_{10}$ -byte RAM is used, it would probably be assigned addresses  $0000_{16}$  through  $01FF_{16}$ . When any one of these addresses appear on the address bus, the RAM is selected via its chip enable line. Notice that a portion of the address bus connects directly to the RAM. This selects the individual byte within the RAM.

In the same way, the ROM is assigned a range of addresses. If a  $1024_{10}$  byte ROM is used, it may be assigned addresses  $FC00_{16}$  through  $FFFF_{16}$ . The ROM must be enabled whenever any of these addresses appear on the address bus. The output latch and input buffers are also assigned unique addresses. Thus, the MPU can communicate with any one of the external circuits simply by placing the proper address on the address bus.

The second problem is more fundamental. It arises because of the basic 2-state nature of digital logic circuits. Recall that the output of a standard logic gate will always be either logic 1 (high) or logic 0 (low). The problem is: Which state should the outputs of the circuits that are connected to the data bus assume when they are not selected? Regardless of which state they assume, they interfere with the output of the enabled circuit. For example, if the output of the disabled circuits assume a high state, they interfere with the low output of the enabled circuit. In other words, one circuit tries to pull the bus line high while the other is trying to force it low.

In the past, this problem has been overcome by using gates with open collector outputs. While open collector devices could be used to solve this problem in microprocessors, an entirely different approach is most often used. To understand how this problem is overcome, we must discuss 3-state logic.

### 3-State Logic



As the name implies, 3-state logic devices have a unique third state in addition to the normal 1 and 0 output. Figure 7-3 compares a standard non-inverting buffer with a 3-state, non-inverting buffer.

Recall that a non-inverting buffer increases the current drive of the input signal without changing the logic levels in any way. Thus, the output may be able to drive ten times as many gates as the input. The standard buffer has one input and one output. The output always assumes the same logic level as the input. Because the input must be either 1 or 0, the output must be the same.

Figure 7-3

Comparison of standard and 3-state buffers.

By contrast, the 3-state buffer has two inputs. In addition to the normal data input, the buffer has an enable/disable input. This input may be either logic 1 or logic 0 depending upon whether we wish to enable or disable the buffer. The buffer shown in Figure 7-3B is enabled by applying logic 1 to the enable/disable input.



When enabled, the 3-state buffer acts exactly like the standard buffer. The output will assume the same logic level as the data input.

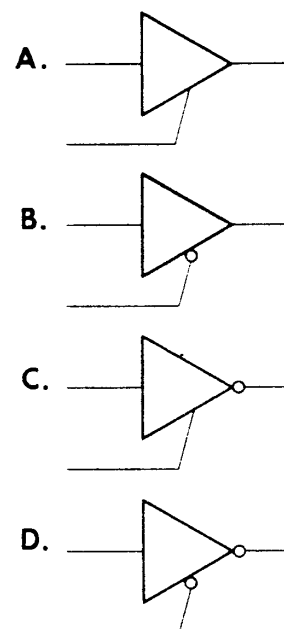
The 3-state buffer is disabled by applying logic 0 to the enable/disable input. When disabled, the output assumes a very high impedance state that is neither logic 1 nor logic 0. While in this high impedance state, the output can be assumed to be disconnected from the rest of the circuit. That is, when the buffer is disabled, its output will not interfere with the circuits to which it is connected.

There are many different types of 3-state devices available. Figure 7-4 shows four different types of 3-state buffers. The buffer shown in Figure 7-4A is the same as that described above. It does not invert and is enabled by logic 1. Notice that the buffer shown in Figure 7-4B has a small circle at the enable/disable input. This means that the buffer is enabled by a logic 0 and disabled by a logic 1.

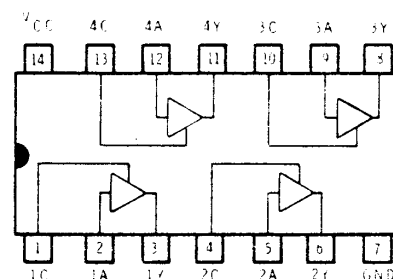
Figures 7-4C and D show inverting buffers. The first is enabled by a logic 1; the second is enabled by a logic 0.

Generally, four or more 3-state buffers are included in a single integrated circuit. Figure 7-5 shows the 74126 type TTL IC. It contains four 3-state buffers in a single 14-pin dual-in-line package.

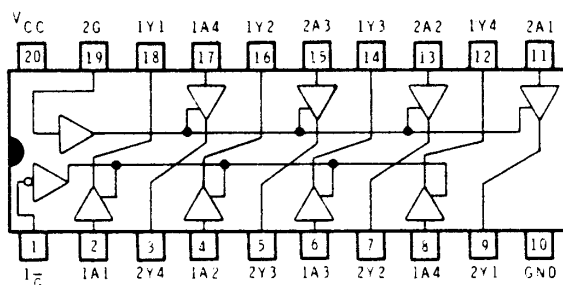
Figure 7-6 shows eight 3-state buffers in a single 20-pin package. The four lower buffers are enabled by a logic 0 at pin 1 of the IC. The four upper buffers are enabled by a logic 1 at pin 19. The input buffer shown earlier in Figures 7-1 and 7-2 could use this type of IC. Buffers of this type are often called bus extenders, line drivers, line receivers, etc., depending on how they are used.



**Figure 7-4**  
Four types of 3-state buffers.



**Figure 7-5**  
The 74126 IC contains four 3-state buffers in a single 14-pin package.



**Figure 7-6**  
The 74LS241 contains eight 3-state non-inverting buffers.

While many different forms of 3-state buffers are available, many microprocessor support circuits do not require separate 3-state buffers. Most RAMS and ROMS have their own 3-state buffers built in. Thus, any time the RAM or ROM is not selected, it automatically goes to its third state. In this state, the outputs are said to be off, disconnected, disabled, floating, or in their high impedance state.

## The 6800 and 6808 MPU Interface Lines

Before you can interface any microprocessor to its support circuits or to the outside world, you must become familiar with its pin assignments, control lines, etc. Figure 7-7 shows the pin assignments of the 6800 and 6808 MPU. Notice that the MPU is in a single 40-pin dual-in-line package.

Most of the pin assignments, as shown in Figure 7-7, are common to both the 6800 and 6808 microprocessors. There are six pin assignments that are not common, pin 3 and 35 through 39. The functions in parenthesis are for the 6808. The functions on these same pins that have no parenthesis are for the 6800 microprocessor. In the following description of the MPU interface lines, those common to both the 6800 and 6808 will be described first. Then those that apply to the individual MPU will be discussed under a separate heading.

Notice that pins 9 through 20 and 22 through 25 make up the 16-bit address bus. These pins connect to 16 three-state output drivers in the MPU. Each driver is capable of driving one standard TTL load. When disabled, the address lines act as open circuits. This capability is sometimes used to allow another device to gain control of the address bus. In this way, some external device can address memory. This is referred to as "direct memory access" (DMA).

Pins 26 through 33 are used for the 8-bit data bus. This is a bi-directional bus that is used to transfer data to and from memory and the input-output circuitry.

You are already familiar with four of the control lines: the read/write line, the reset line, the non-maskable interrupt, and the interrupt request line. The read/write ( $R/\bar{W}$ ) line tells the peripheral devices and memory whether the MPU is in the read or write mode. A read operation is indicated by a logic 1 on this line. In this mode, the MPU reads data from memory or from an input device. A write operation is indicated by a logic 0 on the  $R/\bar{W}$  line. In this mode, the MPU sends data out to memory or an output device. Since it works hand-in-hand with the address bus, the  $R/\bar{W}$  line has a 3-state capability.

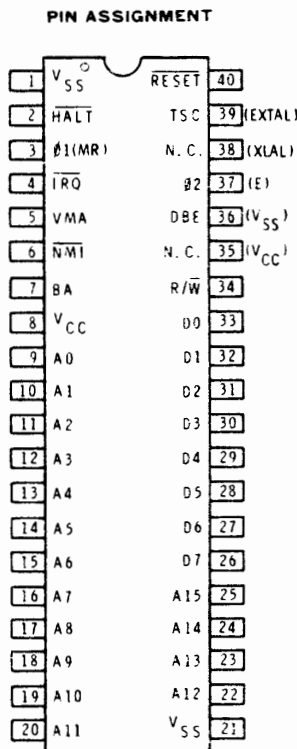


Figure 7-7  
The 6800 and 6808 MPU  
pin assignments.

The reset line (pin 40) was discussed in the previous unit. Recall that it is used to reset and start the MPU when power is initially applied or at anytime that we wish to initialize the system. When this line goes to logic level 1, the MPU starts the reset sequence. Recall that the reset vector is retrieved from addresses  $FFFE_{16}$  and  $FFFF_{16}$ . This vector is loaded into the program counter so that the first instruction is fetched from that address. This capability is used to direct the MPU to the start of the monitor or control program.

The non-maskable interrupt ( $\overline{NMI}$ ) was mentioned when interrupts were discussed. A high-to-low transition on pin 6 (the  $\overline{NMI}$  line) will initiate the non-maskable interrupt sequence. Recall that this forces the MPU to pick up the  $\overline{NMI}$  vector at addresses  $FFFC$  and  $FFFD$ . This vector is the address of the  $\overline{NMI}$  service routine.

The interrupt request line ( $\overline{IRQ}$ ) was also discussed earlier. While this is similar to the non-maskable interrupt, there are three fundamental differences. First, the  $\overline{IRQ}$  signal is maskable; it will be ignored if the interrupt mask bit is set. Second, the  $\overline{IRQ}$  sequence picks up the  $\overline{IRQ}$  vector from addresses  $FFF8_{16}$  and  $FFF9_{16}$ . Third, the  $\overline{IRQ}$  line (pin 4) is level sensitive. That is, a logic 0 on this pin causes the interrupt sequence. Compare this to the  $\overline{NMI}$  line which requires a logic 1-to-0 transition.

The valid memory address (VMA) line is an output. It indicates to peripheral devices that the address on the address bus is a valid one. This signal is necessary because occasionally a false address will appear on the address bus. The VMA signal is used to disable peripheral devices when the address is not valid. A logic 1 at pin 5 indicates that the address is valid and that the peripheral devices may respond accordingly. A logic 0 indicates that the address is not valid and that the peripheral devices should ignore the address. As you will see later, the VMA line is used in any decoding scheme.

The  $\overline{HALT}$  line (pin 2) provides a hardware method of halting the MPU. If this input is forced low, the MPU will finish its present instruction, then it will halt. When halted, all 3-state lines go to their **off** state. This effectively disconnects the MPU from the address and data buses. This line is sometimes used to implement single instruction operation. By controlling the  $\overline{HALT}$  line, the MPU can be forced to stop after each instruction is executed. This can be a valuable aid in troubleshooting hardware and debugging programs. In many applications, the halt capability is simply not required. In this case, the  $\overline{HALT}$  line is permanently connected to +5 volts.

The final control line that is common to both the 6800 and 6808 microprocessors in identity and function is the **bus available** (BA) line. This output (pin 7) indicates whether or not the MPU is executing instructions. Recall that the MPU may stop executing instructions for either of two reasons. First, the WAI instruction will cause the MPU to stop until an interrupt is received. Or, the MPU can be stopped by forcing the  $\overline{\text{HALT}}$  line low. A logic 0 on the bus available line indicates that the MPU is running. A logic 1 indicates that the MPU has stopped. When the MPU is stopped, all 3-state outputs go to their off state. Thus, the MPU is effectively disconnected from the buses. The BA signal indicates that this condition exists by going to the logic 1 state. During this period the buses are available for other purposes such as DMA operations.

### THE 6800 MPU INTERFACE LINES

While you are not yet familiar with the remaining 10 pins of the 6800 MPU, five of them are self-explanatory. For example, two of them (pins 35 and 38) are not connected. Also, three of the pins are used for power. Pin 8 is labeled  $V_{cc}$ . This is the +5-volt supply. Pins 1 and 21 are labeled  $V_{ss}$ . These are ground pins. Notice that the 6800 MPU requires a single +5-volt supply.

The remaining five lines require a brief explanation. First, there are two clock signals labeled  $\phi 1$  (pronounced "phi one") and  $\phi 2$ . A 2-phase non-overlapping clock is required. This must be provided to the MPU from some external clock generator. The details of the clock signal will be discussed later. The normal clock frequency is 1 MHz, although higher speed versions of the 6800 MPU are also available.

The TSC line (pin 39) is used to enable or disable the 3-state address bus and the R/W line, which also has a 3-state capability. This input to the MPU is used in applications in which some external device must periodically take over the address bus. Normally, the MPU has complete control of the address bus and read/write line. However, an external device can effectively disconnect the MPU from the address bus by switching the TSC line to the high state. When TSC goes high, the address bus and read/write line of the MPU go to their off or high impedance state. This allows the external device direct access to memory without going through the MPU. This is called direct memory access or DMA. In many applications, DMA operations are not required and TSC is permanently connected to ground.

The *data bus enable (DBE) line* (pin 36) is the 3-state control line for the data bus. If this input to the MPU is forced low, the data bus will switch to its off or high impedance state. As you will see later, all data transfers between the MPU and memory take place when the  $\phi 2$  clock is high. For this reason, the DBE line is often connected to the  $\phi 2$  clock.

### THE 6808 MPU INTERFACE LINES

The 6808 microprocessor has a total of five pins for power. Pins 8 and 35 are labeled  $V_{cc}$ . This is the +5-volt supply. Pins 1, 21, and 36 are labeled  $V_{ss}$ . These are ground pins.

The 6808 has an internal oscillator that may be crystal controlled. The crystal is connected between pins 38 and 39. These are designated as **XLAL** and **EXTAL** on Figure 7-7. A divide-by-four circuit has been fabricated within the 6808 so that a 4 MHz crystal connected to pins 38 and 39 result in a 1 MHz output. This output (pin 37) is designated as “E”. The 1 MHz output is similar to the  $\phi 2$  signal on the 6800 microprocessor. The Enable (E output) supplies the clock for peripheral devices.

The final control line of the 6808 that we will consider is the **memory ready (MR)**. This input (pin 3) is a control signal which allows stretching of the 1 MHz E output. When MR is high, E will be a normal pulse width. When MR is low, E may be stretched multiples of half periods. This feature allows for interfacing to slow memories.



## Instruction Timing

Before going further, the timing relationship between the various control and bus signals will be discussed. The discussion will start with the most basic timing relationship: the timing for a single instruction. The following discussion applies to both the 6800 and 6808 microprocessors. The only significant difference is where the  $\phi 1$  and  $\phi 2$  clocks are generated: internal to the 6808, external to the 6800. You should also keep in mind that  $\phi 2$  in the following discussion is the E output of the 6808.

Figure 7-8 shows the 2-phase clock signals required by the microprocessor. These two clock signals control every single action that takes place in the MPU and its peripheral devices. To illustrate this, the significant events that occur during the fetch and execution of the LDAA immediate instruction will be discussed. Recall that this is a 2-byte instruction. The first byte is the opcode ( $86_{16}$ ). The second byte is the number that is to be loaded into accumulator A. This instruction requires two MPU cycles. During the first cycle, the opcode is fetched and decoded. During the second cycle, the operand is retrieved from memory and is placed in accumulator A.

The significant events are illustrated. Notice that the events occur at the edges of the clock pulses. Assume that, prior to time 1, the program counter contains the address of the LDAA immediate instruction.

At time 1, the address is transferred from the program counter to the address bus via the memory address register. Notice that this occurs at the positive-going edge of the  $\phi 1$  clock. If the VMA and  $R/\overline{W}$  lines are not already at logic 1, they will be switched to logic 1. A logic 1 on VMA indicates to memory that this is a valid address. A logic 1 on  $R/\overline{W}$  indicates to memory that the MPU wishes to read the byte at the indicated address.

Time 2 is the falling edge of the  $\phi 1$  clock. At this time, the program counter will be incremented by one to the address of the next byte in memory. However, this will not change the address on the address bus. Remember that this address is latched into the memory address register. Thus, the output address is still that of the first byte of the LDAA instruction.

The events which occur during the  $\phi 1$  clock are initiated from within the MPU itself. In fact, in most systems, the  $\phi 1$  clock is applied *only* to the MPU. The  $\phi 2$  clock, on the other hand, is applied to the peripheral circuits as well as the MPU. Thus, the RAMs, ROMs, etc., are controlled by the  $\phi 2$  clock.

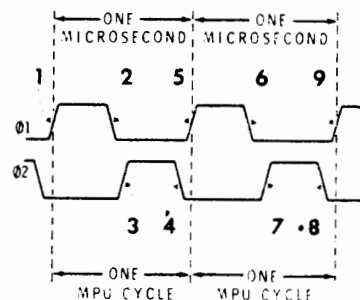


Figure 7-8.

Timing for the LDAA immediate instruction.

Time 3 is the rising edge of the  $\phi 2$  clock. This positive-going edge forces memory to place the data from the indicated address onto the data bus. Recall that this is the opcode for the LDAA immediate instruction or  $86_{16}$ . Notice that the address has had from time 1 to time 3 to stabilize.

Time 4 is the falling edge of the  $\phi 2$  clock. At this time, the MPU accepts the byte from the data bus. Notice that the data bus has from time 3 to time 4 to stabilize. The MPU transfers this byte ( $86_{16}$ ) to the instruction register. There, it is decoded and is interpreted as an LDAA immediate opcode. This tells the MPU that the next byte in memory is the operand that is to be loaded into accumulator A.

This completes the first MPU cycle. During this cycle, the opcode was simply read from memory and decoded. This corresponds to the fetch phase discussed earlier for our hypothetical MPU. Notice that an MPU cycle corresponds to one cycle of the clock. Now let's see what happens during the second cycle or execute phase of the instruction.

At time 5, the address of the operand is transferred from the program counter to the address bus. At time 6, the program counter is incremented by one in anticipation of the next fetch phase.

At time 7, the rising edge of the  $\phi 2$  clock causes the operand to be transferred from memory to the data bus. At time 8, the operand is latched into the MPU where it is transferred to accumulator A. This completes the second MPU cycle and the execution phase of the instruction. Time 9 represents the start of the fetch phase for the next instruction. The LDAA immediate instruction required two MPU cycles or two cycles of the clock. Assuming a 1 MHz clock, two microseconds are required for this instruction.



## Timing of Program Segment

Now that you are familiar with the timing of a single instruction, several instructions will be put together to form a sample program. You can then study the timing relationships between the bus signals, clock signals, the  $R/\overline{W}$  line, etc.

Our sample program segment is shown in Figure 7-9. Using the immediate addressing mode, it loads  $07_{16}$  into accumulator A and adds  $21_{16}$ . The result is then stored in location  $0001_{16}$ . Notice that the first instruction resides at address  $0010_{16}$ .

HEX ADDRESS	HEX CONTENTS	MNEMONIC/ HEX CONTENTS	COMMENTS
0010	86	LDAA#	Load Accumulator A immediate with
0011	07	07	07.
0012	8B	ADDA#	Add to Accumulator A immediate
0013	21	21	21.
0014	97	STAA	Store the result
0015	01	01	at this address.
0016	...	...	Next instruction

Figure 7-9  
Sample program segment.

Figure 7-10 illustrates the timing relationships. At the top the  $\phi 1$  and  $\phi 2$  clock signals are shown. The information that appears on the buses and control lines for each clock period is shown at the bottom. This program segment requires eight clock or MPU cycles. These are numbered one through eight. Next, you will see what happens during each of these cycles.

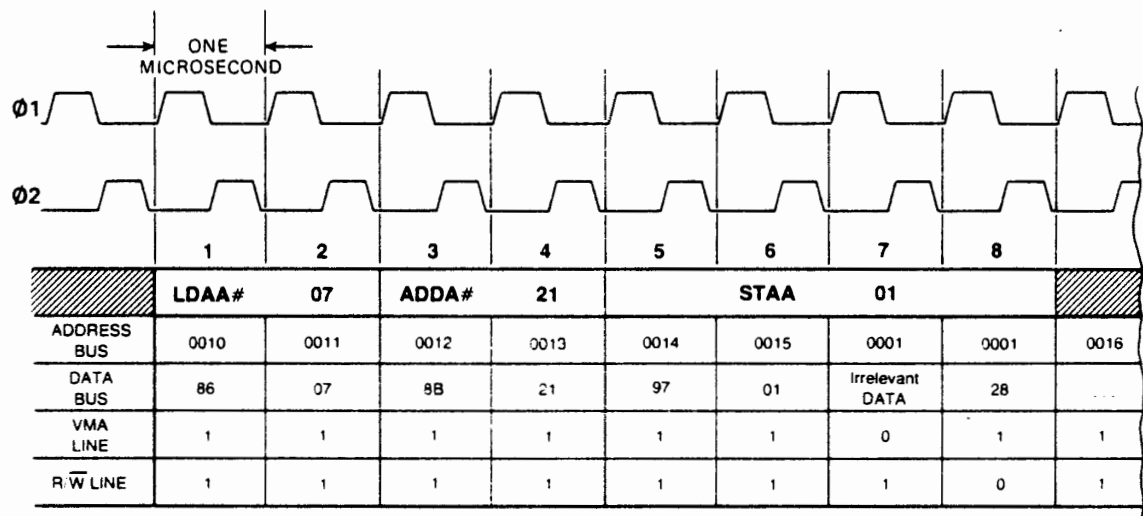


Figure 7-10

Timing of sample program segment.

**Cycle 1.** During the first cycle, the address of the LDAA# instruction ( $0010_{16}$ ) is placed on the address bus. As a result, the opcode  $86_{16}$  is read from that address and is picked up by the MPU. Since this was a read operation from a valid address, the VMA and  $R/\overline{W}$  lines are at logic 1. The MPU decodes the opcode and recognizes that this is an LDAA# instruction. Consequently, it knows that the next byte in memory is the operand that is to be loaded into the accumulator. During this cycle, the program counter was incremented to  $0011_{16}$  so that it now points at the operand.

**Cycle 2.** This is the execution phase of the LDAA# instruction. The address of the operand ( $0011_{16}$ ) is placed on the address bus. The operand ( $07_{16}$ ) is read out on the data bus and is placed in accumulator A. In the process, the program counter is incremented to  $0012_{16}$ . This completes the execute phase of the first instruction.

**Cycle 3.** This is the fetch phase of the next instruction. The address  $0012_{16}$  is placed on the address bus. The opcode at that address is read out and placed on the data bus. The MPU picks up the opcode, decodes it, and discovers that it is an ADDA# instruction. In the process, the program counter is incremented to  $0013_{16}$ .

**Cycle 4.** Here, the address  $0013_{16}$  is transferred to the address bus and the selected memory location is read out. Therefore, the operand  $21_{16}$  is placed on the data bus. The operand is picked up by the MPU and is added to the contents of the accumulator. The sum  $28_{16}$  is retained in accumulator A. The program counter is incremented to  $0014_{16}$ .

**Cycle 5.** This is the fetch phase for the third instruction. The address  $0014_{16}$  is placed on the address bus. The opcode for STAA is read out and decoded. The MPU recognizes that the direct address mode is used. Thus, it will interpret the next byte in memory as the address at which the sum is to be stored. The program counter is incremented to  $0015_{16}$ .

**Cycle 6.** Address  $0015_{16}$  is placed on the address bus. Notice that the contents of this location is the address at which the sum is to be stored. Thus, 01 is read out on the data bus where it is picked up by the MPU. Because the MPU recognized that direct addressing is indicated, it assumes that the address at which the sum is to be stored is  $0001_{16}$ . This address is retained in the MPU. The program counter is incremented to  $0016_{16}$ .

**Cycle 7.** During this cycle, the MPU prepares to store the sum. To do this, it must transfer the address  $0001_{16}$  to the address register. Also, it must transfer the sum from the accumulator to the data register. While this is happening, the MPU must refrain from all external data transfers. To prevent unwanted data transfers, the MPU switches the VMA line low. This tells all peripheral devices that the address is not a valid one and that no data transfers should be initiated. Thus, the peripheral devices simply ignore the data and address buses for this cycle.

**Cycle 8.** The MPU is now ready to store the sum in memory. The address at which the sum is to be stored ( $0001_{16}$ ) is placed on the address bus. The data to be stored ( $28_{16}$ ) is placed on the data bus. The VMA line is switched high, indicating that this is a valid address. The  $R/\overline{W}$  line is switched low, indicating that this is a store operation. Thus, the sum ( $28_{16}$ ) is stored away in memory location  $0001_{16}$ .

Of course, the computer does not stop here. During the next cycle, the next instruction is fetched and decoded. However, the eight machine cycles illustrated above should give you the idea. As you will see later, the timing relationships shown here become important when we interface the MPU with memory or I/O circuitry.

## The 6800 and 6808 Data Sheets

The preceding section has given you some information on the control lines and timing relationships of both microprocessors, but you may have some questions that have not been answered here. For this reason, a detailed data sheet on these microprocessors is included in Appendix B of this course. It explains in more technical language the capabilities of the microprocessors. At this point in your study, you should have little trouble understanding these data sheets. You may want to refer to these data sheets if you have a question that is not answered in the text.

## Self-Test Review

1. What is a 3-state logic gate?
2. How is the non-inverting buffer shown in Figure 7-4A switched to its high impedance state?
3. In the 6800 MPU, how can the address bus be forced to the off state?
4. How can the data bus be forced to its off state?
5. What does a logic 0 on the VMA line indicate?
6. What does a logic 1 on the BA line indicate?
7. How can the MPU be stopped by hardware?
8. How does the MPU indicate that it wishes to store data in memory?
9. List three ways in which an interrupt request differs from a non-maskable interrupt.
10. The following is a list of the MPU's buses and control lines. Characterize each as either input, output, bidirectional, or internally generated.

$\overline{\text{Halt}}$	_____	R/ $\overline{\text{W}}$	_____
$\phi 1$	6800 _____ 6808 _____	DBE	_____
$\overline{\text{IRQ}}$	_____	$\phi 2$	_____
VMA	_____	TSC	_____
BA	_____	$\overline{\text{NMI}}$	_____
Address Bus	_____	$\overline{\text{Reset}}$	_____
Data Bus	_____	E	_____

11. In reference to the clock signals, when is a new address placed on the address bus?
12. When is the program counter incremented?
13. When does memory place data on the address bus?
14. When does the MPU pick up or latch in data that is on the data bus?

## Answers

1. A gate which has an off state in addition to the normal logic 1 and logic 0 states.
2. By applying a logic 0 to the enable/disable input.
3. By applying a logic 1 to the 3-state control (TSC) line.
4. By applying a logic 0 to the data bus enable (DBE) line.
5. That the address is not valid and should be ignored.
6. That the MPU has halted.
7. By applying a logic level 0 to the  $\overline{\text{HALT}}$  input.
8. By switching the  $\text{R}/\overline{\text{W}}$  line to low.
9. First, the interrupt request is maskable;  $\overline{\text{NMI}}$  is not.  
 Second,  $\overline{\text{IRQ}}$  is level sensitive;  $\overline{\text{NMI}}$  is edge sensitive.  
 Third,  $\overline{\text{IRQ}}$  uses the interrupt vector at address  $\text{FFF8}_{16}$  and  $\text{FFF9}_{16}$ , while  $\overline{\text{NMI}}$  uses the vector at  $\text{FFFC}_{16}$  and  $\text{FFFD}_{16}$ .
10.
 

HALT	—	<u>input</u>	R/W	—	<u>output</u>
$\phi 1$	6800	<u>input</u>	6808	<u>internal</u>	DBE — <u>input</u>
IRQ	—	<u>input</u>	$\phi 2$	—	<u>input</u>
VMA	—	<u>output</u>	TSC	—	<u>input</u>
BA	—	<u>output</u>	NMI	—	<u>input</u>
Address Bus	—	<u>output</u>	Reset	—	<u>input</u>
Data Bus	—	<u>bi-directional</u>	E	—	<u>output, internal</u>
11. On the rising edge of the  $\phi 1$  clock.
12. On the falling edge of the  $\phi 1$  clock.
13. On the rising edge of the  $\phi 2$  clock.
14. On the falling edge of the  $\phi 2$  clock.

## INTERFACING WITH RANDOM ACCESS MEMORY

The Cassette tape segment entitled "Semiconductor Memories" is an excellent overview of the types of memories encountered in microprocessor applications. If you have not already done so, you should complete this audio-visual activity at this time.

As shown in the audio-visual presentation, there are several different types of semiconductor memories. Today, the most popular type of memory used with microprocessors is the static RAM.

Recall that a RAM is a random access read/write memory. The static RAM uses bistable flip-flops to store data. Because the data is latched in these flip-flops, no refresh circuitry is required. That is, data can be maintained indefinitely without refreshing as long as power is applied.

Many different types of static RAMs are available. The smaller static RAMs use the bipolar TTL technique. Most of the larger RAMs use either MOS or CMOS technology. One of the most popular sizes of static RAMs is 1024 bits. This size RAM is packaged in a single IC having from 16 to 24 pins. Internally, the 1024 memory cells may be arranged as 128 8-bit words. Thus, a microprocessor system that requires 512 bytes of RAM would require four of these IC's. IC's of this type require eight pins as data lines. For this reason, 24-pin packages are often used. Since price is related to package size, a RAM of this type is often more expensive than a RAM that uses a smaller package.

A more popular scheme is to arrange the 1024 memory cells into 256 four-bit words. With this arrangement, only four data pins are required. Since the microprocessor works with 8-bit words, two of these RAMs must be used to form a 256 by 8-bit memory. As before, four packages are required to form the 512-byte memory. However, the four packages tend to be smaller and probably less expensive.

Another popular arrangement is the 1024 by 1-bit RAM. This scheme uses only one data line per package. Of course, eight of these packages must be used to form 8-bit bytes. Also, a problem arises if we wish to have a memory smaller than 1024 by 8.

## The Static Ram Storage Cell

The basic storage cell for an MOS type static RAM is shown in Figure 7-11. Keep in mind that this cell stores a single bit of data. There may be 1024 of these along with address decoders and bus drivers in a single IC.

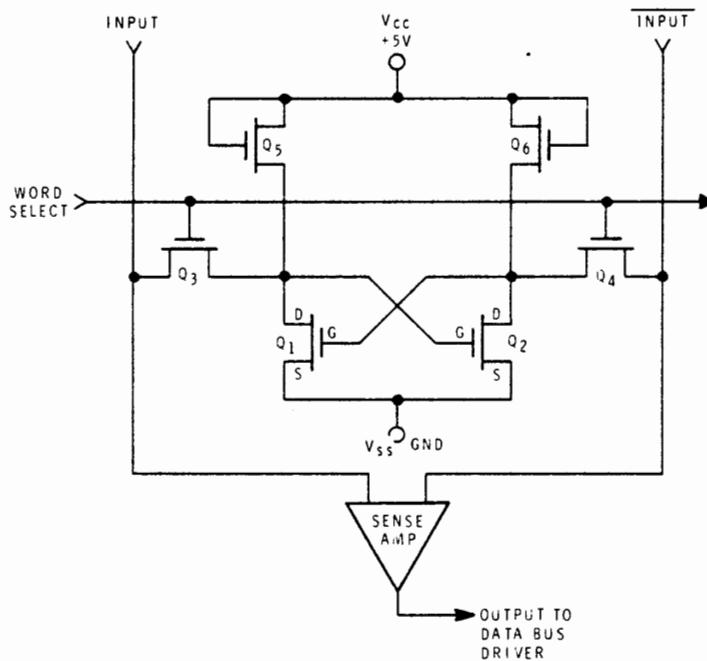


Figure 7-11  
The static RAM storage cell.

The storage cell itself consists of the six MOS transistors. In keeping with current convention, the simplified symbol for the MOS transistor is used. Assume that these are N-channel enhancement type MOSFETs. Recall that an enhancement type MOSFET is normally off or non-conducting. However, since these are N-channel devices, they can be made to conduct by placing a positive voltage on the gate terminal. That is, when the gate is near ground potential, the transistor is off and represents a very high impedance. But, when the gate is high (near  $+V_{CC}$ ), the transistor conducts and has a much lower impedance. If you keep these points in mind, the operation of the cell is easy to understand.

Transistors  $Q_1$  and  $Q_2$  are cross coupled so that they form a bistable latch or flip-flop. The bit of data is stored in this latch.  $Q_5$  and  $Q_6$  act as load resistors for  $Q_1$  and  $Q_2$  respectively. MOSFETs are used instead of physical resistors because they take up less chip space.  $Q_3$  and  $Q_4$  act as switches which connect input data to the latch during write operations. Also, they connect the data in the latch to the output sense amplifier during read operations. The word select line is connected to the gates of  $Q_3$  and  $Q_4$ . A logic 1 on the word select line will turn  $Q_3$  and  $Q_4$  on. A logic 0 will keep them turned off.

Since this is a RAM storage cell, we must be able to write into it and to read from it. The following discussion will show how we write into it.

**Write Operation.** Before writing a bit of data into the cell, the discussion will first define what constitutes a 1 and 0. Assume that the latch contains a binary 1 when  $Q_2$  conducts and  $Q_1$  is cut off. And, of course, it contains a binary 0 when  $Q_1$  conducts and  $Q_2$  is cut off.

When power is initially applied, the flip-flop will latch in one condition or the other but we can never be sure of which. We write data into the cell by controlling the input, input, and word select lines. To store a binary 1, we place a logic 1 (a positive voltage) at the input line. The input line is always the opposite state of the input line, so it will go to logic 0 ( $\approx 0$  volts). The binary 1 is now stored by momentarily applying a logic 1 (positive pulse) to the word select line.

A positive pulse on the word select line will turn switches  $Q_3$  and  $Q_4$  on. Thus, the positive voltage on the input line is applied through  $Q_3$  to the gate of  $Q_2$ . This forces  $Q_2$  to conduct. When  $Q_2$  conducts, its drain voltage falls to a low value. This reduced voltage is felt on the gate of  $Q_1$ , cutting  $Q_1$  off. When  $Q_1$  cuts off, its drain voltage goes high. This increased voltage is felt on the gate of  $Q_2$ , holding  $Q_2$  in the on state.



When the word select line returns to logic 0,  $Q_3$  and  $Q_4$  cut off. However, the conduction of  $Q_2$  keeps  $Q_1$  cut off and the high drain voltage of  $Q_1$  keeps  $Q_2$  conducting. Thus, the binary 1 is latched in the flip-flop. It will remain there until power is removed or until a binary 0 is deliberately written in.

We can later write in a 0 if we like by applying a logic 0 to the input, a logic 1 to  $\overline{\text{input}}$  and then pulsing the word select line. When the word select line goes high,  $Q_4$  conducts, applying the logic 1 from  $\overline{\text{input}}$  to the gate of  $Q_1$ . This tends to bring  $Q_1$  out of cutoff. At the same time,  $Q_3$  conducts, applying the logic 0 from the input to the gate of  $Q_2$ . This tends to cut off  $Q_2$ . As you can see, the flip-flop latches in the opposite state. This represents a binary 0.

The sense amplifier and output line (shown at the bottom) are disabled during this period by the read/write line (not shown).

**Read Operation.** The input and  $\overline{\text{input}}$  lines are 3-state lines that are enabled or disabled by the read/write line. When the read/write line (not shown) is high, the input and  $\overline{\text{input}}$  lines are effectively disconnected. During this period, we can read data from the storage cell by pulsing the word select line. Assume that the cell is presently storing a logic 1. This means that  $Q_1$  is cut off and that  $Q_2$  is conducting. Consequently  $Q_1$ 's drain voltage is high (logic 1) while  $Q_2$ 's drain voltage is low. When the word select line swings high,  $Q_3$  conducts, connecting the high voltage at the drain of  $Q_1$  to the left input of the sense amplifier. At the same time,  $Q_4$  conducts, connecting the low voltage at the drain of  $Q_2$  to the right input of the sense amplifier. The sense amplifier interprets this as a logic 1 condition and sets the output line accordingly.

If the flip-flop contains a logic 0 when the word select pulse occurs, the right input of the sense amplifier receives the higher voltage while the left input receives the lower. The sense amplifier interprets this as a logic 0.

## A 128-Word by 8-Bit Ram

Because a storage cell can store only one bit of information, large numbers of these cells are required to form useful memory sizes. Figure 7-12 shows how 1024 cells can be arranged to form a 128-word by 8-bit RAM. Each of the squares represents one of the 6-transistor cells just discussed.

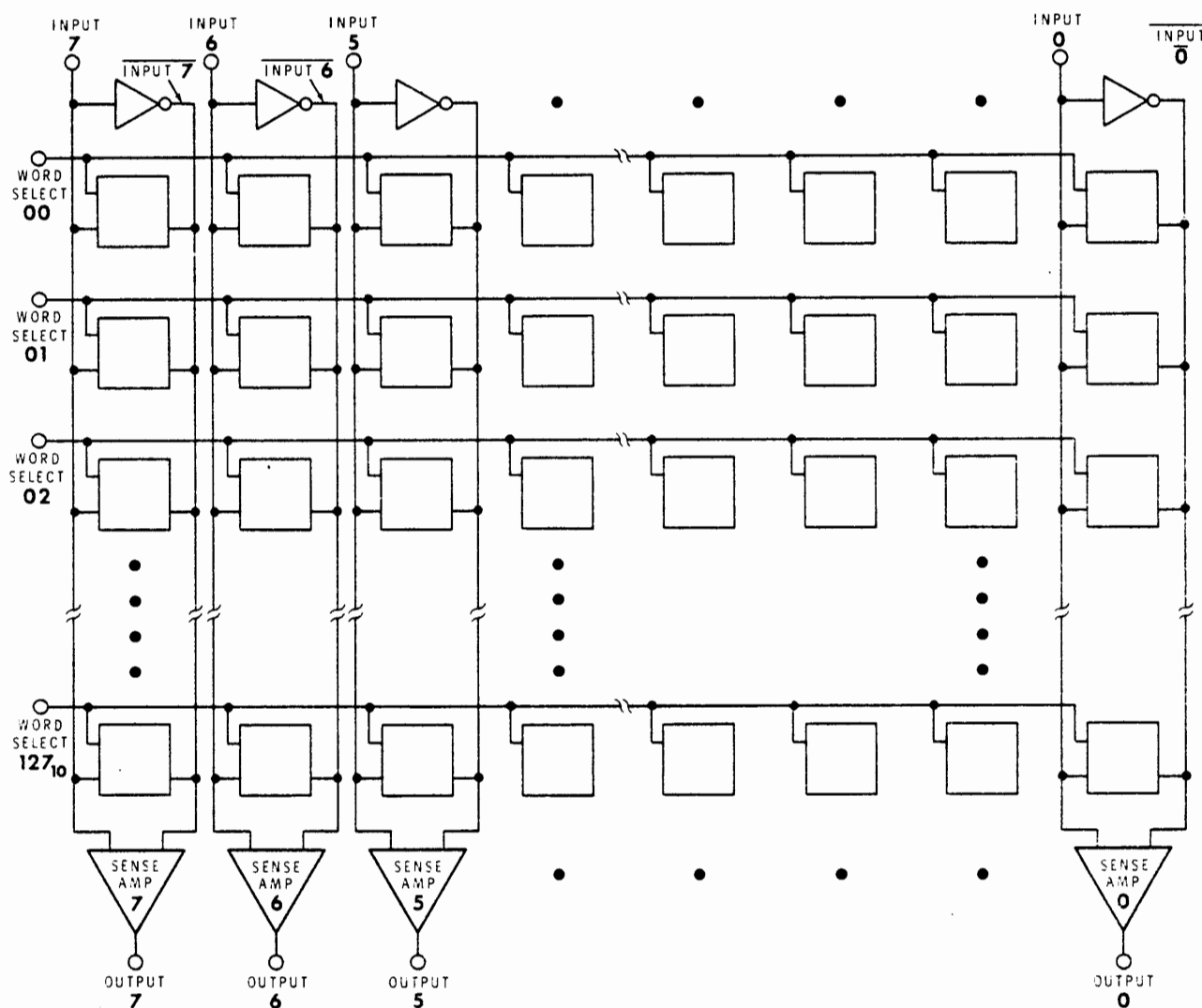
The word select lines are shown entering on the left. While only four are shown, there would actually be 128 of these lines — one for each word. Notice that word select line 00 connects to each of eight storage cells across the top of the figure. In an actual system, these eight storage cells might make up the 8-bit byte we call memory location  $0000_{16}$ .

The input lines are shown at the top of the diagram. For simplicity, only four of the eight lines are shown. Notice that each input line is inverted so that complement inputs can be applied to each cell. Although the details are not shown, both the input and input lines are 3-state lines so that they are effectively disconnected except during a write operation.

The output lines are shown at the bottom of the diagram. These lines are disabled during a write operation but they are enabled during a read operation. One sense amplifier is required for each output line.

Keep in mind that Figure 7-12 does not show the complete RAM. It merely shows the memory storage matrix and the sense amplifiers. Some additional circuits are required to turn this into a working RAM. One of these is an address decoder.

The memory array is arranged as 128 bytes. An address decoder is required that can select any of these 128, 8-bit storage locations. Thus, a 1 of 128 decoder is required. The input to the decoder is the seven address lines from the MPU. Recall that seven bits can specify 128 different addresses.



**Figure 7-12**

Here 1024 storage cells are arranged to form a 128-byte by 8-bit RAM.

The address decoder is made up of 128 7-input AND gates. A simplified diagram is shown in Figure 7-13. Here, only three gates are shown — the first two and the last. Word select line 00 should go high when address lines  $A_0$  through  $A_6$  are all low. Notice that seven inverters are used to form  $\overline{A_0}$  through  $\overline{A_6}$ . Notice also, that these complements are the inputs to the top AND gate. If  $A_0$  through  $A_6$  are all low, then  $\overline{A_0}$  through  $\overline{A_6}$  must be all high. Consequently, the output of the AND gate goes high. Thus, word select line 00 is selected when the low order address is  $0000000_2$ .

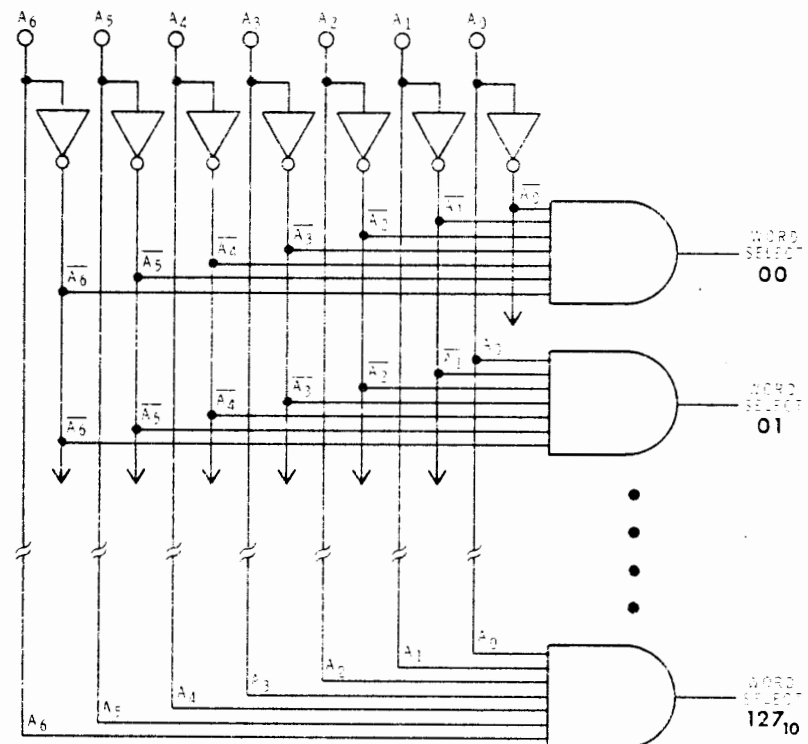


Figure 7-13

The 1 of 128 address decoder.

The 01 select line is activated when the address is  $0000001_2$ . The next  $125_{10}$  gates are not shown. Word select line  $127_{10}$  is selected when  $A_0$  through  $A_6$  are all high. Thus, it is activated when the address is  $1111111_2$ .

Most RAMs do not have separate input and output lines. Instead they have data lines which can serve either as inputs or outputs. This is possible because the MPU cannot read and write data simultaneously.

Figure 7-14 shows a simplified arrangement. The data lines are shown on the left. The 3-state input buffers are enabled by a high signal on the WRITE line. This line is controlled by the  $R/\overline{W}$  signal and the chip enable (CE) signal. As you will see, the WRITE line is high when the MPU is writing data into memory. This enables the input buffers and allows data to be written into the selected address. The output buffers are disabled during this period by the low signal on the READ line.

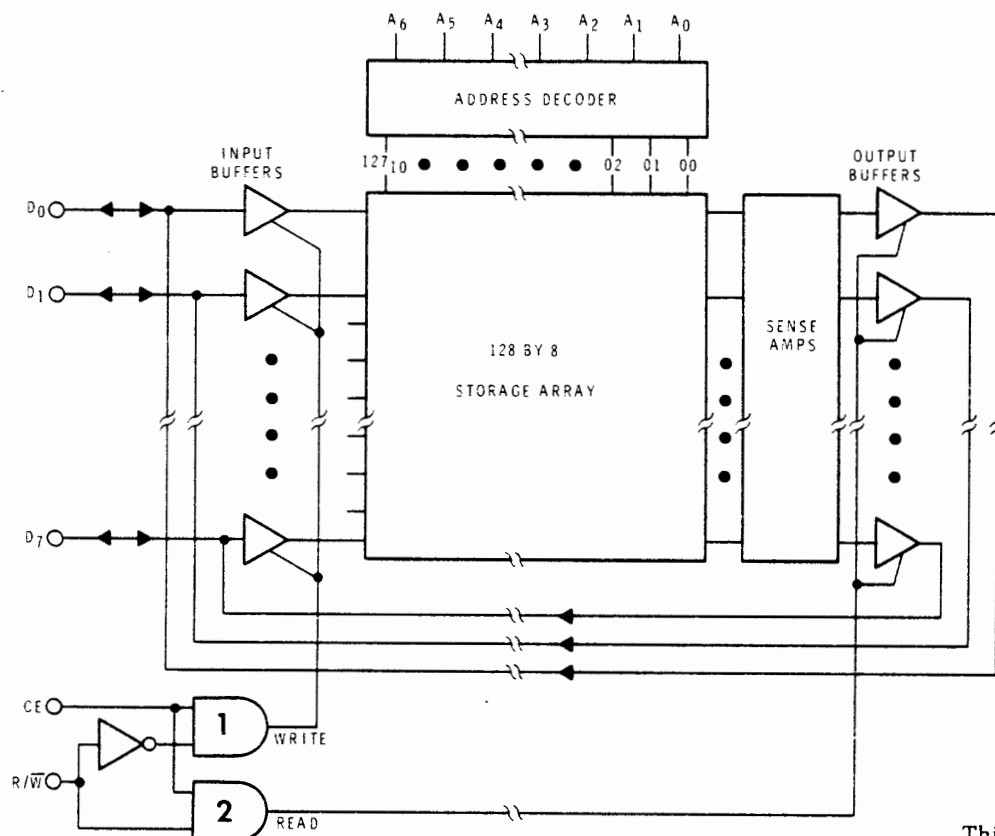


Figure 7-14

This RAM has bi-directional data lines.

When data is to be read from the RAM, the READ line is switched high and the WRITE line is switched low. This disables the input buffers and enables the output buffers. Thus, the data at the selected address is read out and placed on the data bus.

As Figure 7-14 illustrates, the READ and WRITE signals are controlled by the chip enable (CE) signal and the  $R/\overline{W}$  line. The CE input line is switched high when this particular memory chip is selected. If this line is low, gates 1 and 2 are disabled. This causes both the READ and WRITE signals to go low, disabling both the input and output buffers. In effect, it disconnects this chip from the data bus.

However, when  $\overline{CE}$  is high, the  $R/\overline{W}$  line controls the READ and WRITE signals. This  $R/\overline{W}$  line is connected to the  $R/\overline{W}$  line of the MPU. Recall that the  $R/\overline{W}$  line is low when the MPU is writing data into RAM, and high when the MPU is reading from RAM. When  $R/\overline{W}$  is high, the output of gate 2 goes high, enabling the output buffers. The output of gate 1 is held low, disabling the input buffers. This places the RAM in the read mode.

When  $R/\overline{W}$  goes low, the output of gate 1 goes high and the output of gate 2 goes low. This places the RAM in the write mode.

Some RAMs have a single chip enable line. In many RAMs, the chip enable line is labeled  $\overline{CE}$ , meaning that the chip is selected when the enable line is low. Some RAMs have several chip enable (CE) or chip select (CS) lines.

Figure 7-15 shows a 128 by 8 RAM that is designed to be used with the 6800 MPU. As shown in the simplified block diagram, this RAM has six chip select lines. The large number of chip selects allows this RAM to be used with little or no external address decoding. As shown in the pin assignment diagram, a 24-pin package is required. This RAM is called the 6810. Data sheets on this device are included in Appendix B of this course.

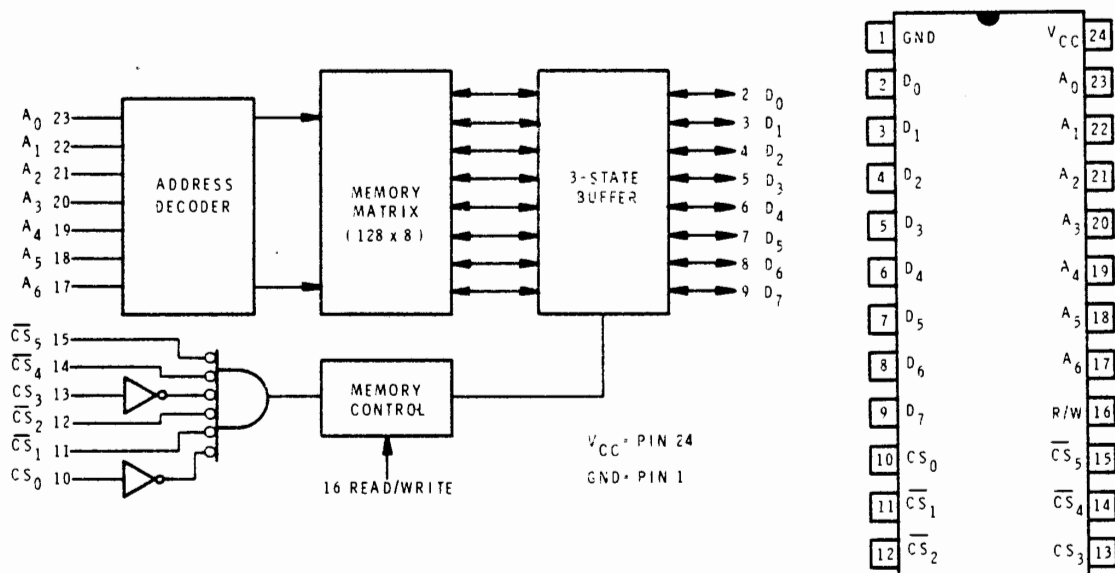


Figure 7-15

The 6810 is a 128-byte by 8-bit RAM.

## A 256 by 4-Bit Ram

Figure 7-16 shows the block diagram of a popular 256 by 4-bit static RAM. In 8-bit systems, two of these would be required to form a 256<sub>10</sub> byte memory. Like the 128<sub>10</sub> by 8 RAM, this circuit has 1024<sub>10</sub> storage cells. To simplify the address decoders, the cells are arranged in 32 rows of 32 cells each. The 32 cells in each row are further divided into 8 columns of 4 bits each. Thus, the array consists of 32 rows by 8 columns by 4 bits (or 256 by 4 bits).

Two address decoders are used. The row select decoder is a 1-of-32 decoder which chooses the row. Five address lines ( $A_0$  through  $A_4$ ) are used to specify the proper row.

A 1-of-8 column decoder is used to select the proper column. Three address lines ( $A_5$  through  $A_7$ ) are used to specify the proper column. The selected 4-bit word is at the point where the row and column lines intersect.

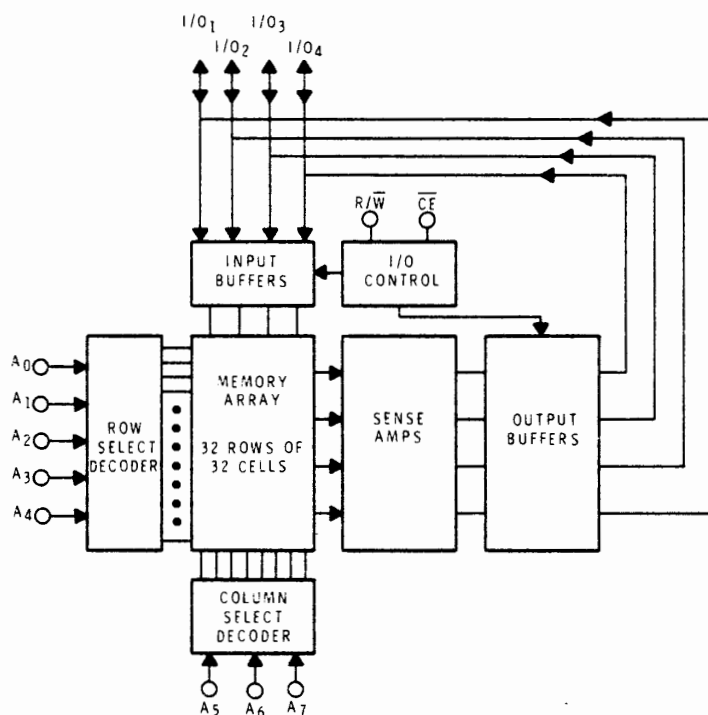


Figure 7-16  
Block Diagram of a 256 by 4 RAM.

In this particular IC, the data lines are called I/O lines. The  $R/\overline{W}$  line determines whether the IC is in the read or write mode. The  $\overline{CE}$  line determines if this particular IC has been selected. If you count the pins shown and add one for  $V_{cc}$  and another for ground, you will see that a 16-pin package is required. Figure 7-17 shows the pin assignments and logic diagram of the popular 2112 static RAM. It has the same general arrangement as that shown in the block diagram.

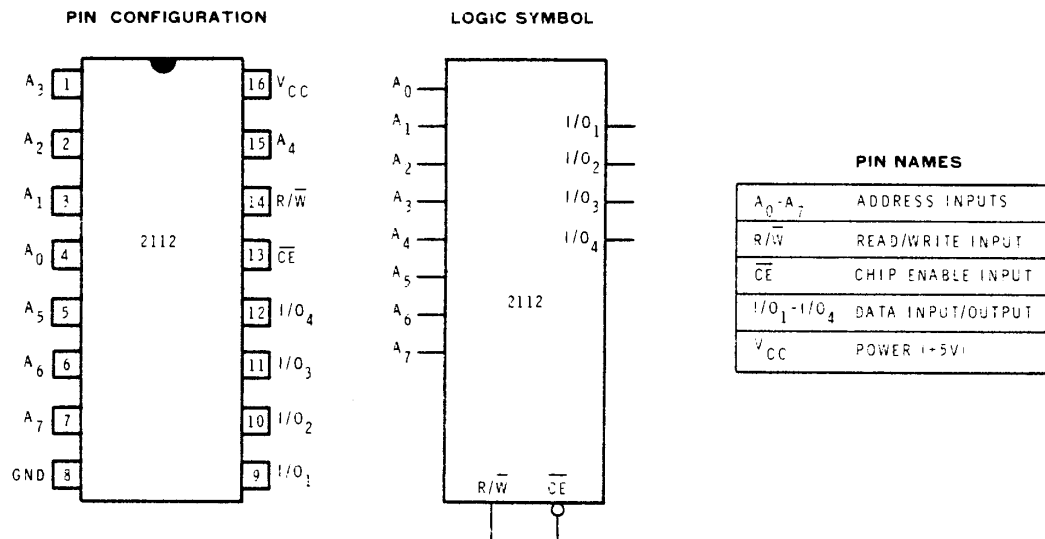


Figure 7-17

Pin assignments and logic symbol for the 2112 static RAM.

## Connecting Ram to the MPU

Figure 7-18 is a partial schematic of a 6800 based microprocessor system. Two 2112 IC's are used as a 256<sub>10</sub>-by-8 RAM. Although not shown, the address and data buses from the MPU are also connected to a ROM, input and output circuits, and possibly additional RAMs. The MPU can communicate with only one of these devices at any one time. To communicate with the two 2112 ICs, the MPU must first select them by switching their  $\overline{CE}$  lines low. Notice that the  $\overline{CE}$  lines are connected to the output of the RAM address decoder. This decoder monitors the eight high-order address lines. When the address of the RAM appears on these lines, the two 2112 chips are enabled.



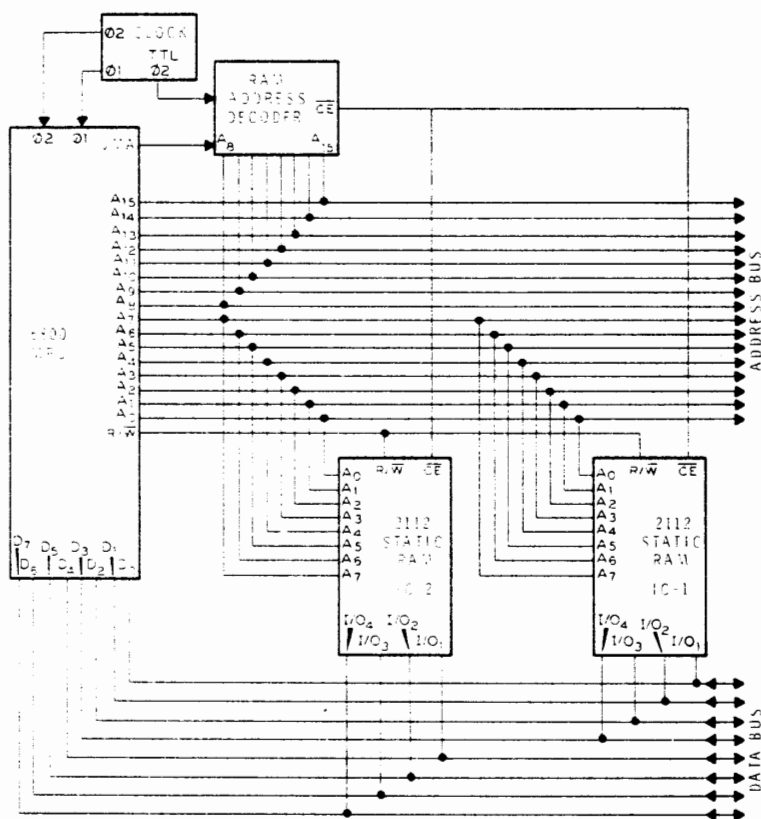


Figure 7-18

Partial schematic of a 6800 based microprocessor using two 2112 static RAMs.

The two 2112 IC's make up a 256 by 8-bit RAM. IC1 connects to the four least significant bits of the data bus (D<sub>0</sub> through D<sub>3</sub>), while IC2 connects to the four most significant bits. Thus, when a byte of data is stored in the RAM, the four LSB's go in IC1, while the four MSB's go in IC2. This is possible because the two IC's are enabled at the same time. Notice that the address,  $\overline{CE}$ , and  $R/\overline{W}$  lines of the two ICs are tied together.

The address lines of the IC's monitor the A<sub>0</sub> through A<sub>7</sub> lines from the MPU. Once the chips are enabled, any one of the 256 memory locations can be selected by placing the proper address on the lower eight address lines.

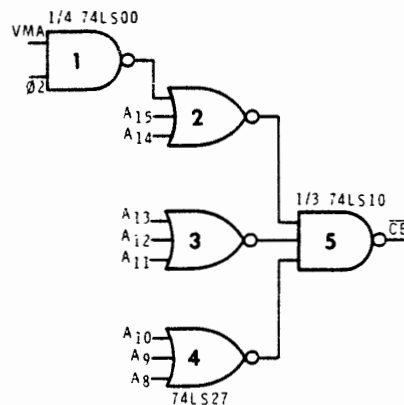
## Address Decoding

The  $256_{10}$ -byte RAM must be assigned some starting address. In 6800-based systems, a common practice is to assign RAM the lowest addresses. Thus, a  $256_{10}$ -byte RAM would be given address  $0000_{16}$  through  $00FF_{16}$ . In binary, these are addresses  $0000\ 0000\ 0000\ 0000_2$  through  $0000\ 0000\ 1111\ 1111_2$ . Notice that the eight LSB's can specify any one of the  $256_{10}$  memory locations. However, it is the upper eight bits that specify that the starting address is at the low end of memory. That is, the RAM must be enabled when the upper eight bits of the address bus are  $0000\ 0000$ .

In Figure 7-18, the RAM address decoder monitors the upper eight bits of the address bus. When this decoder finds that the upper eight bits are all zeros, it switches the  $\overline{CE}$  line low, enabling the RAM. However, notice that VMA and the  $\phi 2$  clock signals are also applied to the decoder. Recall that VMA is 1 when the address is valid. Thus, the decoder must also monitor the VMA line, since the RAM should not be enabled unless the address is valid. Recall also that the MPU must receive and transmit data only while the  $\phi 2$  clock is at logic 1. Thus, the decoder also monitors the  $\phi 2$  clock. If the high-order address lines are all zeros, the VMA line is high, and the  $\phi 2$  clock is high, the RAM will be enabled.

The address decoder can be any type of logic circuit that meets the above requirements. A typical circuit is shown in Figure 7-19. If you trace through the various logic levels, you will see that  $\overline{CE}$  is low only when  $A_8$  through  $A_{15}$  are low and VMA and  $\phi 2$  are high. NAND gate 1 produces a low at its output when VMA and  $\phi 2$  are high. The other inputs to NOR gate 2 and the inputs to NOR gates 3 and 4 are low when the high-order address is  $0000\ 0000_2$ . The three NOR gates produce high outputs. Thus, the inputs to NAND gate 5 are all high. This forces the output of gate 5 low. As you can see, this circuit fulfills the requirements of the address decoder.

Figure 7-19  
Address decoder using  
discrete logic gates.



A memory location in the RAM shown in Figure 7-18 is selected by all 16 address lines. The low-order address lines connect directly to the RAM IC's, while the high-order lines connect to the RAM address decoder. Thus, each memory location in RAM has only one address. The addresses are said to be *fully decoded*.

We can save some decoding logic by only *partially decoding* the address. Figure 7-20 shows an address decoder that monitors only two of the address lines ( $A_{14}$  and  $A_{15}$ ). If you trace the logic levels through, you will see that  $\overline{CE}$  is low any time that  $A_{14}$  and  $A_{15}$  are low and VMA and  $\phi 2$  are high.  $A_{14}$  and  $A_{15}$  will be low for any address at or below  $3FFF_{16}$ . If the  $\overline{CE}$  line is used to enable a RAM that has fewer than  $3FFF_{16}$  bytes, some of the addresses will be duplicated. To illustrate this point, assume that we replace the address decoder shown in Figure 7-18 with the circuit shown in Figure 7-20. Memory location  $0000_{16}$  can be selected by placing address  $0000_{16}$  on the address bus. However, location  $0000_{16}$  is also selected when  $0900_{16}$  is placed on the address bus. The reason for this is that the RAM is enabled because  $A_{14}$  and  $A_{15}$  are low. And, the lowest address in the RAM is read out because  $A_6$  through  $A_7$  are low. Actually, there are dozens of addresses that will select any given location. For example, addresses  $3F00_{16}$ ,  $2700_{16}$ ,  $1C00_{16}$ , and many others will all select memory location  $0000_{16}$ . This is the price that must be paid for saving a few gates in the address decoder.

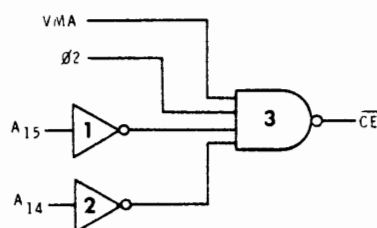


Figure 7-20

We can save gates by only partially decoding the address.

In practice, this does not actually cause many problems. All we have done is sacrifice the lower  $16,384_{10}$  addresses to  $256_{10}$  bytes of RAM. However, this still leaves three-fourths of the 65 K of addresses untouched. In most cases, this leaves more than enough addresses for I/O devices, ROMs, etc.

The fact that a given byte of data appears at several addresses is also no problem as long as we remember this limitation in our programming. Partial decoding schemes are frequently used because they save decoding logic.

## Self-Test Review

15. List three common arrangements for a 1024-bit static RAM.
16. Using 256 by 4-bit static RAMs, what is the smallest 8-bit RAM possible?
17. What is the advantage of using two 256 by 4-bit RAMs to form 256 by 8-bit memory rather than using two 128 by 8-bit RAMs.
18. How many bits of data can be stored in the circuit shown in Figure 7-11?
19. Explain how data is written into this circuit.
20. Explain how data is read from this circuit.
21. Refer to Figure 7-13. Which address lines would connect to the input of the AND gate that drives word select line 02?
22. Refer to Figure 7-14. What is the state of the input and output buffers if CE is low?
23. When CE is high and  $\overline{R/\overline{W}}$  is low, the circuit shown in Figure 7-14 is in the \_\_\_\_\_ state.
24. In addition to some address lines, what other signals are connected to an address decoder in a 6800-based microcomputer system?
25. How is the 2112 static RAM enabled?
26. Refer to Figure 7-19. What conditions must be met before  $\overline{CE}$  will go low?
27. What is the advantage and disadvantage of only partially decoding an address?
28. When each byte of memory has one and only one address, the memory is said to be \_\_\_\_\_ decoded.

## Answers

15. The most common arrangements for a 1024-bit static RAM are:

1024 by 1-bit  
256 by 4-bits  
128 by 8-bits

16. 256 by 8-bits.

17. Because the 128 by 8-bit RAM requires 8 data lines, it uses a larger and usually more expensive package.

18. Only one at a time.

19. The input data bit sets the INPUT and  $\overline{\text{INPUT}}$  lines to the proper complementary states, then the word select line is pulsed high. This turns  $Q_3$  and  $Q_4$  on connecting the inputs to the flip-flop. This sets or resets the flip-flop to the proper state.

20. The two input lines are tri-stated. The word select line is pulsed high turning on  $Q_3$  and  $Q_4$ . This connects the outputs of the flip-flop to the sense amplifier. In turn, the sense amplifier sets the data line to the proper state.

21.  $\overline{A}_0$ ,  $A_1$ ,  $\overline{A}_2$ ,  $\overline{A}_3$ ,  $\overline{A}_4$ ,  $\overline{A}_5$ , and  $\overline{A}_6$ .

22. Both the input and output buffers are disabled.

23. Write.

24. The VMA line and the  $\phi 2$  clock signal.

25. The 2112 static RAM is enabled by applying a logic 0 to the  $\overline{\text{CE}}$  line.

26. VMA and  $\phi 2$  must be high. At the same time, address lines  $A_8$  through  $A_{15}$  must be low.

27. The advantage is that generally fewer logic gates are required. The disadvantages are that it wastes addresses and that a given byte of data may be accessed at several different addresses.

28. Fully.

## INTERFACING WITH DISPLAYS

One of the most popular output devices used with the microprocessor is the 7-segment LED display. It can be used to display the decimal digits 0 through 9 or the hexadecimal digits 0 through F. It can also display many special characters. Its low cost and flexibility make the 7-segment LED display ideal for low cost microprocessor based systems.

### The 7-Segment Display

The 7-segment display consists of seven LED's arranged in the pattern shown in Figure 7-21A. An eighth LED is generally included to act as a decimal point. By lighting the LED's in different combinations, 256<sub>10</sub> patterns are possible. These range from a blank display (all segments off) to the digit 8 with a decimal point (all segments on). Figure 7-21B shows some of the patterns that can be formed.

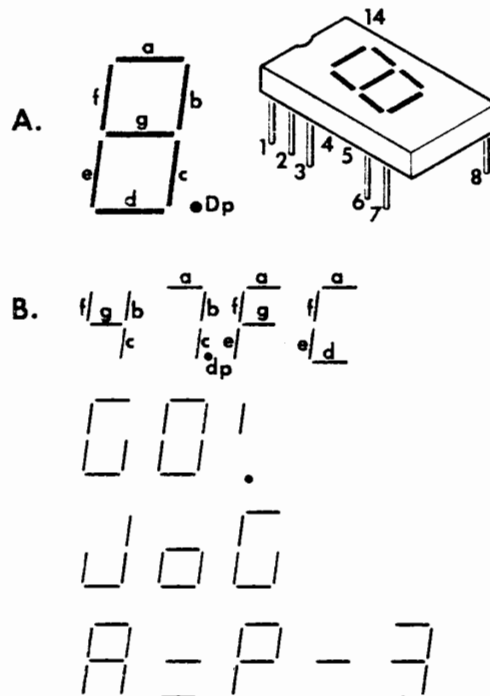


Figure 7-21

The 7-segment display can be used to form many numerals, letters, and symbols.

Two types of 7-segment displays are popular. One is called the common-anode type. As shown in Figure 7-22A, the anodes of the eight light-emitting diodes are tied together and connected to  $+V_{cc}$ . A diode is forward biased (turned on) by applying a low logic level to its cathode. An external series resistor is required to reduce the current to an acceptable level.

The other type is called the common-cathode type. As Figure 7-22B illustrates, the cathodes are tied together and connected to ground. In this case, a diode is forward biased (turned on) by applying a high logic level to its anode.

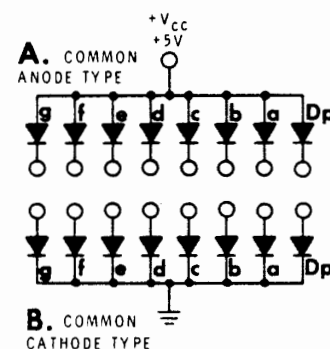


Figure 7-22  
Two types of 7-segment displays.

## Driving the 7-Segment Display

A number of IC's are especially designed to drive the 7-segment display. A typical example is the 7447 shown in Figure 7-23A. This is a BCD-to-7-segment decoder-driver. It receives a 4-bit binary number at inputs A, B, C, and D. It provides the proper patterns at outputs a through g to form the numerals 0 through 9 and six special characters as shown in Figure 7-23B.

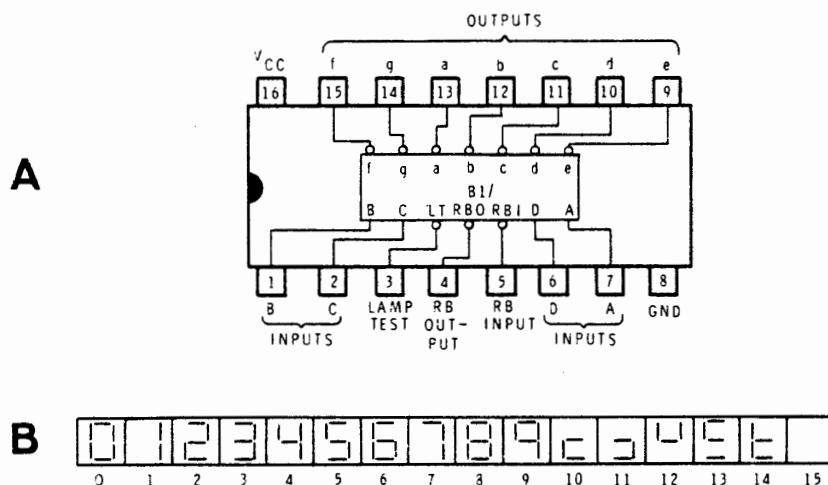


Figure 7-23  
The 7447 seven segment decoder-driver and its resultant displays.

When used with a microprocessor, the MPU does not drive the decoder-driver directly. Recall that the information on the data bus is there for a microsecond or less. Since the 7447 has no latch capability, a separate latch must be used to store the data at the right instant. Figure 7-24 shows a representative circuit.

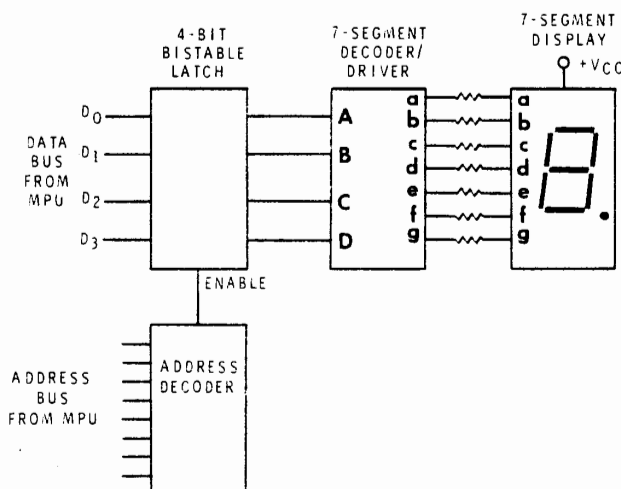


Figure 7-24

Using the decoder/driver.

Here a 4-bit bistable latch is used between the data bus and the decoder-driver. The latch is enabled only at the instant that its address appears on the address bus. For example, assume that the latch has been given the address  $8000_{16}$ . An instruction such as  $STAA\ 8000_{16}$  will activate the display. When this instruction is executed, the address 8000 is placed on the address bus. The address decoder recognizes this address and enables the latch. Thus, the data on lines  $D_0$  through  $D_3$  are latched into the 4-bit latch. An instant later, the MPU places new information on the address and data buses. However, the display responds only to what has been preserved in the 4-bit latch.

The advantage of this circuit is that it requires only one instruction from the MPU to display a given number indefinitely. Its disadvantage is its lack of flexibility. Of the  $256_{10}$  displays possible, only the  $16_{10}$  provided by the decoder-driver can be used. Thus, this arrangement can display only those symbols shown in Figure 7-23B.



A more versatile way of driving a 7-segment display is shown in Figure 7-25. Here, a low current display is used so that the latch can drive the display directly. Notice that the BCD-to-7-segment decoder is eliminated. The seven segments of the display (and the decimal point) are now controlled directly by the eight bits of data on the data bus.

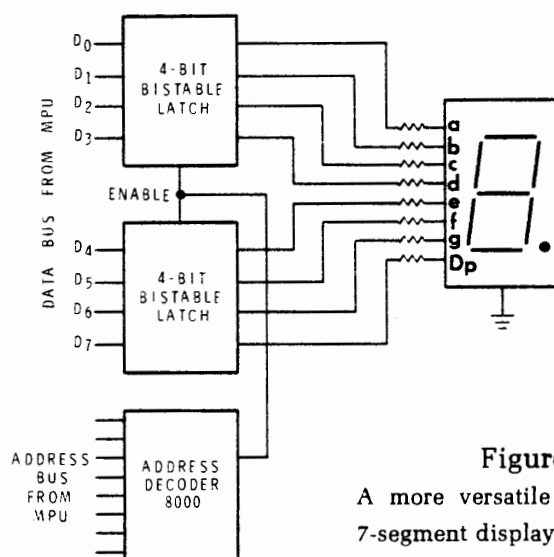


Figure 7-25

A more versatile way of driving a 7-segment display.

The display shown is a common-cathode type. Thus, a segment is turned on by applying a logic high to its input. Figure 7-26 shows the 8-bit pattern that is required to display the digit 1. Since this 8-bit pattern comes from the MPU, the microprocessor must do the decoding. This requires more MPU time and instructions but it greatly increases the versatility of the display. We can now use any of the  $256_{10}$  possible displays by providing the proper 8-bit pattern.

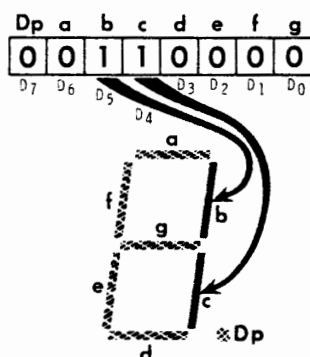


Figure 7-26

Each bit controls one segment of the display.

A display of this type is often used to display the hexadecimal digits 0 through F. The MPU must convert the binary numbers 0000 through 1111 into the proper 8-bit patterns to display the equivalent hexadecimal digit. The easiest way to do this is to use a "look-up" table. The 8-bit patterns are placed in 16<sub>10</sub> consecutive memory locations. Assume that the starting address of the table is FF96<sub>16</sub> as shown in Figure 7-27.

HEX ADDRESS	HEX CONTENTS	MNEMONIC/ CONTENTS	COMMENTS
0010	97	STAA	Store the binary number at the variable offset.
0011	16	16	
0012	CE	LDX#	
0013	FF	FF	
0014	96	96	Load the index register with the starting address of the table.
0015	A6	LDAA,X	
0016	—	variable offset	Load A indexed using the variable offset
0017	B7	STAA	Store the 7-segment code at 8000 <sub>16</sub> .
0018	80	80	
0019	00	00	
D, a b c d e f g			NEXT INSTRUCTION
FF96	7E	0 1 1 1 1 1 1 0	7-Segment pattern for 0
FF97	30	0 0 1 1 0 0 0 0	7-Segment pattern for 1
FF98	6D	0 1 1 0 1 1 0 1	7-Segment pattern for 2
FF99	79	0 1 1 1 1 0 0 1	7-Segment pattern for 3
FF9A	33	0 0 1 1 0 0 1 1	7-Segment pattern for 4
FF9B	5B	0 1 0 1 1 0 1 1	7-Segment pattern for 5
FF9C	5F	0 1 0 1 1 1 1 1	7-Segment pattern for 6
FF9D	70	0 1 1 1 0 0 0 0	7-Segment pattern for 7
FF9E	7F	0 1 1 1 1 1 1 1	7-Segment pattern for 8
FF9F	7B	0 1 1 1 1 0 1 1	7-Segment pattern for 9
FFA0	77	0 1 1 1 0 1 1 1	7-Segment pattern for A
FFA1	1F	0 0 0 1 1 1 1 1	7-Segment pattern for B
FFA2	4E	0 1 0 0 1 1 1 0	7-Segment pattern for C
FFA3	3D	0 0 1 1 1 1 0 1	7-Segment pattern for D
FFA4	4F	0 1 0 0 1 1 1 1	7-Segment pattern for E
FFA5	47	0 1 0 0 0 1 1 1	7-Segment pattern for F

Figure 7-27

Program segment and table for converting binary to 7-segment display format.

A program segment is required that will store the proper 7-segment pattern in the latches shown in Figure 7-25. Assume that the address of the latches is  $8000_{16}$ . The program segment might look like that shown in Figure 7-27.

The program assumes that the binary number we wish to convert to its 7-segment hexadecimal equivalent is in accumulator A. The first instruction stores the binary number at address  $0016_{16}$ . Assume that the number is  $0000\ 0110_2$  or  $06_{16}$ . Thus, this number is stored at address  $0016_{16}$ . Notice that this number becomes the offset for the LDAA, X instruction at address  $0015_{16}$ .

The second instruction loads the starting address of the table ( $FF96_{16}$ ) into the index register. Then, accumulator A is loaded using indexed addressing. The offset, which is now  $06_{16}$ , (by virtue of the first instruction) is added to the contents of the index register ( $FF96_{16}$ ) to form the address of the operand ( $FF9C_{16}$ ). Thus, the number at address  $FF9C_{16}$  is loaded into accumulator A. This number is  $5F_{16}$ , which is the proper 7-segment code to form the numeral 6.

Finally, this number is stored at address  $8000_{16}$ . This is the address of the latches that drive the display. Therefore, if  $06_{16}$  is in accumulator A when this program segment is executed, the digit 6 will be displayed. Also, if  $02_{16}$  is initially in accumulator A, the program segment will cause a 2 to be displayed.

Using this arrangement, two 4-bit latches are required for each display. There are eight bit latches available in a single IC. However, most come in 20 to 24-pin packages, which are relatively expensive. There is one type of 8-bit latch that comes in a 16-pin package. This device will be discussed next.

## Using an Addressable Latch

Figure 7-28 shows the pin outs and schematic for the 8-bit addressable latch. It can store an 8-bit byte and drive a low current display. Its most striking characteristic is that data is entered into the device in serial form. That is, it has a single data input (pin 13). Thus, the eight bits of data must be entered into the device one bit at a time. This explains how the IC can get by with only 16 pins.

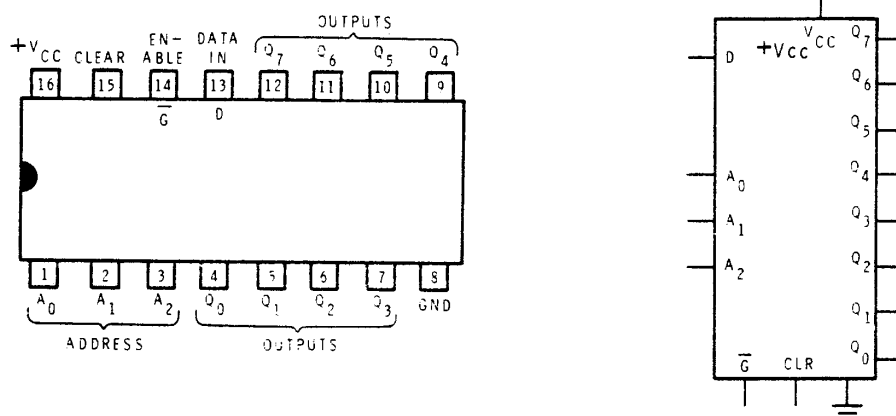


Figure 7-28

Pin outs and schematic diagram of the 74LS259 latch.

The addressable latch has an enable pin that allows it to be selected by an address decoder. A low at pin 14 will enable the latch and allow it to receive data. Actually, there are eight latches on the IC. They are numbered 0 through 7. The particular latch to which the input data bit is routed is determined by the 3-bit address at pins  $A_0$ ,  $A_1$ , and  $A_2$ . Figure 7-29 shows which latch is selected for each address.

ADDRESS INPUTS			LATCH SELECTED
$A_2$	$A_1$	$A_0$	
0	0	0	0
0	0	1	1
0	1	0	2
0	1	1	3
1	0	0	4
1	0	1	5
1	1	0	6
1	1	1	7

Figure 7-29.

Latch selection table.

How can this latch be used by the microprocessor to drive a display? Figure 7-30 shows a representative circuit. The three address lines on the addressable latch are connected to their corresponding lines on the address bus. The remaining lines of the address bus are connected to the address decoder. Assume that the address decoder is arranged so that the addressable latch is enabled for addresses C160<sub>16</sub> through C167<sub>16</sub>. That is, the addressable latch is enabled for eight different addresses. Address C160<sub>16</sub> selects latch 0 of the addressable latch; C161<sub>16</sub> selects latch 1; and so forth.

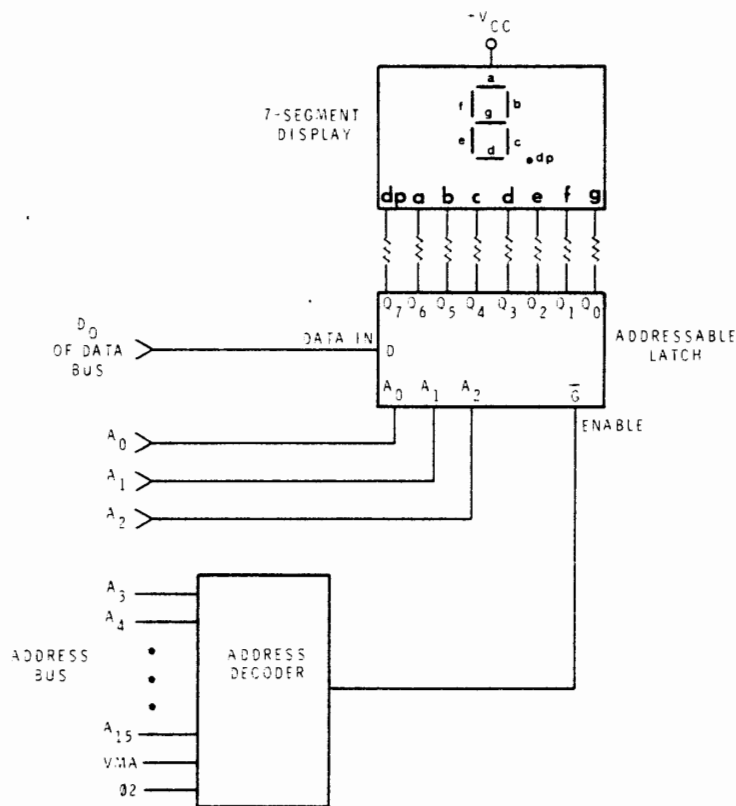


Figure 7-30.

Using the addressable latch to drive a 7-segment display.

The only data line connected to the addressable latch is D<sub>0</sub> from the microprocessor's data bus. The outputs of the eight latches (Q<sub>0</sub> through Q<sub>7</sub>) drive the seven segments and the decimal point of the display.

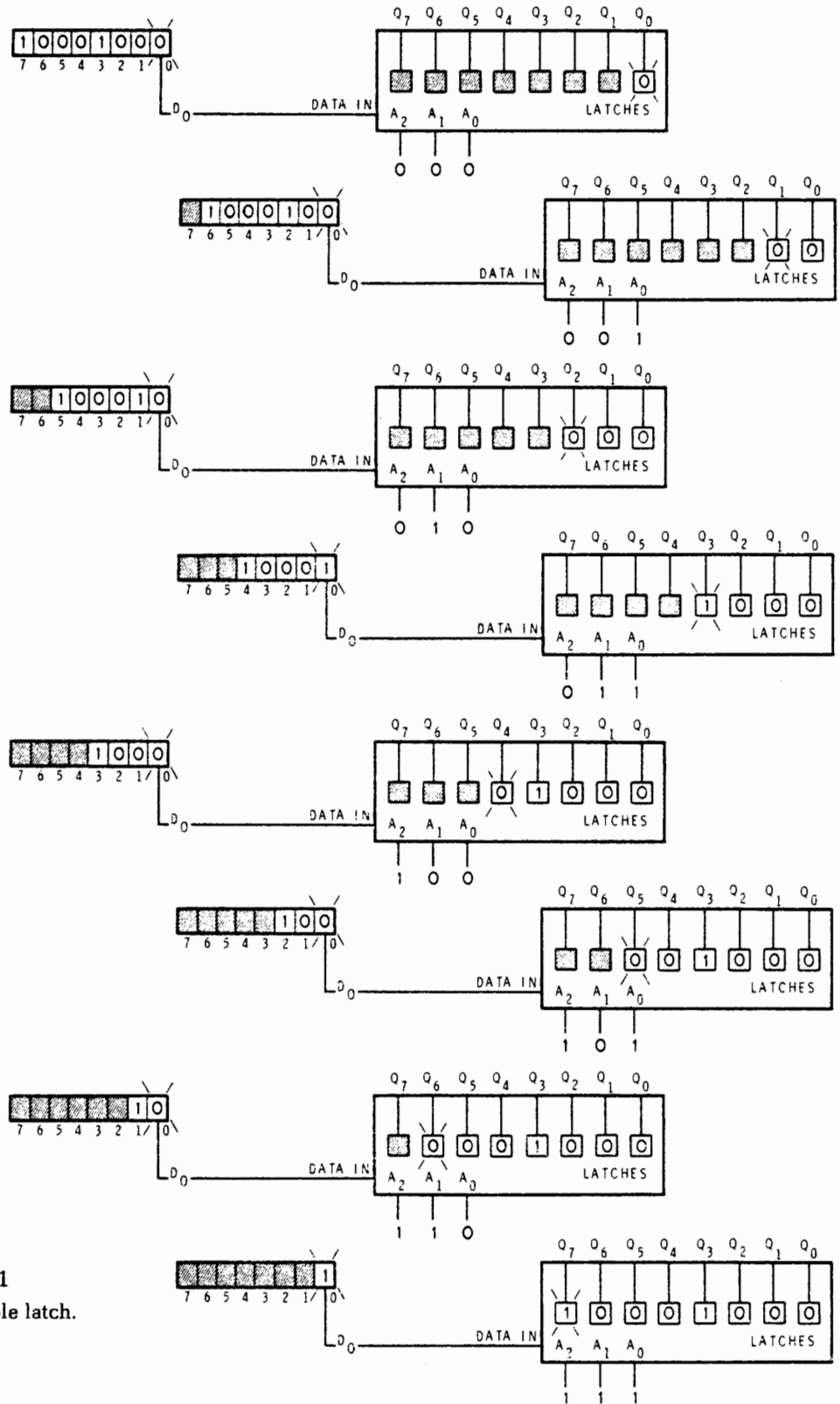


Figure 7-31  
Loading the addressable latch.

While this arrangement results in an inexpensive circuit, it places an extra burden on the MPU. As in the previous example, the microprocessor must make the binary to 7-segment conversion. But in this case, it must also convert the parallel 7-segment code into a serial bit stream that is acceptable to the addressable latch. You are already familiar with the solution to the first problem. Now, the discussion will show how the MPU can convert parallel data to serial data.

The procedure is illustrated in Figure 7-31. The register on the left represents one of the accumulators. It contains the proper 7-segment code for displaying the letter A. This code must be transferred a bit at a time into the addressable latch shown on the right. Remember that the addressable latch actually contains eight latches and that the particular latch selected is determined by the 3-bit address at  $A_0$  through  $A_2$ .

The first step is to store the contents of the accumulator at the basic latch address which is  $C160_{16}$ . If you convert this address to binary, you will find that address bits  $A_0$ ,  $A_1$ ,  $A_2$  are all 0's. Since these lines connect to pins  $A_0$ ,  $A_1$ , and  $A_2$  on the addressable latch, latch 0 is selected as shown. Notice that the only data line used is  $D_0$ . Thus, the 0 in bit 0 of the accumulator is stored in latch 0.

Next, the contents of the accumulator are shifted to the right so that the next bit is available at  $D_0$ . The address of the latch is incremented by 1 and a store instruction is executed. Thus, the second bit is stored in latch 1.

This procedure continues as shown until all eight bits have been shifted from the accumulator to the addressable latch. Of course, a short program segment is required to control this operation. A typical program is shown in Figure 7-32. Step through the program and verify that it works. Remember that the 7-segment code must be in accumulator A before running the program.

The ET-3400 Microprocessor Trainer uses a method similar to this for driving the displays. The program shown in Figure 7-32 will light the left-most display. However, in the ET-3400 Microprocessor Trainer, the  $D_0$  data line is inverted. Therefore, a 1 must be used in the accumulator for each segment that you wish to light. For example, the letter A is formed by placing  $01110111_2$  in accumulator A before you run the program.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0010	CE	LDX #	Load the index register immediate with the address of the latch.
0011	C1	C1	
0012	60	60	Store accumulator A in the latch.
0013	A7	STAA, X	
0014	00	00	Get ready to send next bit.
0015	46	RORA	
0016	08	INX	Set up next address.
0017	9C	CPX#	Compare with final address.
0018	C1	C1	
0019	67	67	If a match does not occur, branch back to here.
001A	26	BNE	
001B	F7	F7	Otherwise, wait.
001C	3E	Wait	

Figure 7-32

A simple program for loading the addressable latch.



## Multiplexing Displays

The previous approaches required one or more latches for each display. There is a method by which we can drive up to eight displays using only two 8-bit latches. However, it requires some additional components and a lot of microprocessor time.

A typical circuit is shown in Figure 7-33. Eight common cathode displays are used in this example. The display cannot light unless its associated transistor is turned on. The transistors are controlled by the contents of the digit select latch. In turn, this latch is loaded by the MPU.  $Q_1$  is turned on by placing 1 in bit 7 of the digit latch;  $Q_2$  is turned on by placing 1 in bit 6; etc. Generally, only one transistor at a time is turned on. A common procedure is to store 10000000 in the digit select latch and rotate the contents so that the 1 appears at each bit in turn.

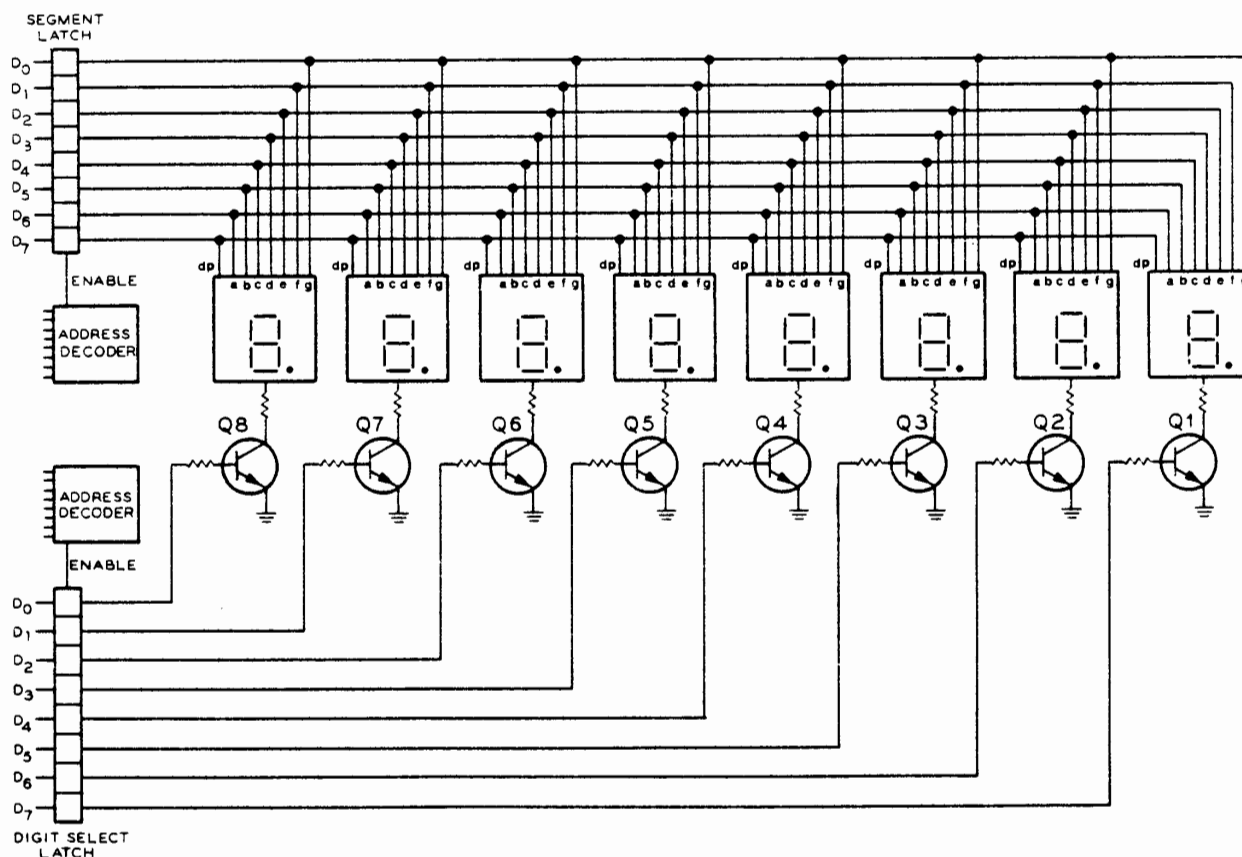


Figure 7-33  
Multiplexing Displays.

The individual segments of each display are turned on by the segment latch. For example, to turn on the decimal point in the right-most display, the segment latch must contain a 1 at bit 7. At the same time, the digit select latch must turn on  $Q_1$ .

To display an 8-digit message, the procedure looks like this: Load the digit select latch with  $10000000_2$ . Load the segment latch with the 7-segment information for the right-most display. This causes the right-most display to show the first digit. After a brief delay, the contents of the digit display latch are changed to  $01000000$ . This turns on the second display. Next the contents of the segment latch must be changed to the 7-segment code for the second display. This procedure continues until all eight displays are lit. Because no two displays are lit simultaneously, the displays must be refreshed many times each second to give the illusion of a constant steady display.

The advantage of this technique is that a minimum amount of hardware is required. However, it requires more MPU time than the previous techniques since the displays must be constantly refreshed.

## Self-Test Review

29. How do we turn on a common-anode type 7-segment display?
30. What is a disadvantage of using a decoder-driver, such as the 7447, to drive a display?
31. Why is a latch required between the MPU and the display?
32. When the latch drives the display directly, what performs the decoding function?
33. List four instructions that could be used to output data to a display.
34. Using the arrangement shown in Figure 7-26, what 8-bit code is required to form the letter P?
35. Refer to the program shown in Figure 7-27. If the number in accumulator A is  $08_{16}$  when this program segment is run, what binary number will be outputted to the display?
36. What is the purpose of the program shown in Figure 7-27?
37. Refer to Figure 7-30. What determines which of the latches the input data bit goes into?
38. The program shown in Figure 7-32 converts \_\_\_\_\_ data to \_\_\_\_\_ data.
39. Refer to Figure 7-33. What determines which character is displayed?
40. Refer to Figure 7-33. What determines the display on which the character appears?

## Answers

- 29. A common-anode type display is turned on by applying logic 0 to the proper cathode.
- 30. The pattern is limited to those provided by the decoder.
- 31. Because the output data is stable for only an instant, a latch must be used to capture the data and hold it for the display.
- 32. The microprocessor.
- 33. Any of the store instructions could be used to output data to a display. These include: STAA, STAB, STX, and STS.
- 34.  $0110\ 0111_2$ .
- 35.  $0111\ 1111_2$ . This is the 7-segment pattern for the numeral 8.
- 36. The program converts binary numbers between 0000 and 1111 to a 7-segment code to form the appropriate hexadecimal numeral.
- 37. The 3-bit address at  $A_0$  through  $A_2$ .
- 38. parallel, serial.
- 39. The number in the segment latch.
- 40. The number in the digit select latch.

## INTERFACING EXPERIMENTS

The hardware experiments are included in Unit 10 of this course. Go to Unit 10 and perform experiments 1 through 4. Some of these experiments are quite involved and you should not attempt more than one experiment per sitting.

## UNIT EXAMINATION

1. A 3-state logic gate:
  - A. Has an enable/disable input as well as the normal data inputs.
  - B. Is effectively disconnected from the circuit when it is disabled.
  - C. Is generally used when two or more gates drive the same bus line.
  - D. All the above.
2. Which of the following lines or buses is bi-directional?
  - A. The read/write line.
  - B. The address bus.
  - C. The data bus.
  - D. All the above.
3. Which of the following lines or buses is an input to the 6800 MPU?
  - A. The address bus.
  - B. The  $\phi 2$  clock.
  - C. Valid Memory Address (VMA).
  - D. Bus available (BA).
4. The MPU can be stopped by external hardware by:
  - A. Forcing the  $\overline{\text{halt}}$  line low.
  - B. Forcing the  $\overline{\text{halt}}$  line high.
  - C. Forcing the BA line low.
  - D. Forcing the BA line high.
5. RAM places data on the data bus when:
  - A. The positive-going edge of the  $\phi 2$  clock occurs.
  - B. The negative-going edge of the  $\phi 2$  clock occurs.
  - C. The positive-going edge of the  $\phi 1$  clock occurs.
  - D. The negative-going edge of the  $\phi 1$  clock occurs.
6. In 6800-based systems, two control lines are generally connected to the address decoder along with the address line. These are:
  - A. BA and  $\phi 1$ .
  - B. VMA and  $\phi 2$ .
  - C. TSC and  $\phi 2$ .
  - D. NMI and  $\phi 1$ .

7. What is the minimum number of 256 by 4 RAM IC's required to develop a memory of 8-bit words?
  - A. One.
  - B. Two.
  - C. Four.
  - D. Eight.
8. The output of the address decoder normally connects to:
  - A. The  $\overline{R/\overline{W}}$  line on the RAM IC's.
  - B. The  $\overline{CE}$  line on the RAM IC's.
  - C. The  $\overline{HALT}$  line of the MPU.
  - D. The  $\overline{NMI}$  line of the MPU
9. What type of circuit is required between the MPU data lines and the 7-segment display?
  - A. A latch.
  - B. A decoder/driver.
  - C. An address decoder.
  - D. All of the above.
10. Refer to the program shown in Figure 7-27. The number in accumulator A when the program starts is used as:
  - A. The address of the corresponding 7-segment code.
  - B. The 7-segment code.
  - C. The offset address for the LDAA, X instruction.
  - D. The address of the 7-segment display.
11. The program shown in Figure 7-32:
  - A. Converts the 7-segment code in accumulator A into a serial bit stream acceptable to the latch.
  - B. Converts the 7-segment code in accumulator A into a parallel data byte acceptable to the addressable latch.
  - C. Converts the binary number in accumulator A into the corresponding 7-segment code.
  - D. Loads eight addressable latches with the proper 7-segment codes to display an 8-character message.
12. An advantage of the circuit shown in Figure 7-33 is:
  - A. It requires fewer MPU instructions than the other methods of driving the display.
  - B. It requires less MPU time than the other methods of driving the displays.
  - C. It requires less external circuitry than the other methods.
  - D. All of the above.



# Individual Learning Program

## MICROPROCESSORS

*Unit 8*

### INTERFACING — PART 2

EE-3401

HEATH COMPANY  
BENTON HARBOR, MICHIGAN 49022

Copyright © 1977  
Heath Company  
*All Rights Reserved*  
Printed in the United States of America

## CONTENTS

Introduction .....	8-3
Unit Objectives .....	8-4
Unit Activity Guide .....	8-5
Interfacing with Switching .....	8-6
The Peripheral Interface Adapter (PIA) .....	8-20
Using the PIA .....	8-33
Interfacing Experiments .....	8-42
Unit Examination .....	8-43
Examination Answers .....	8-45



## *UNIT 8*

# INTERFACING — PART 2

## INTRODUCTION

In this unit, you will continue your study of interfacing the microprocessor with other circuits. The emphasis will be on interfacing with switches and displays, which are the most common input and output devices.

You will also be introduced to a special type of support IC called the peripheral interface adapter. As you will see, this device can greatly simplify many interfacing problems.

As you complete this unit, you will perform several interfacing experiments. As with Unit 7, a basic knowledge of electronics in general and digital techniques in particular is required to gain full benefit from these experiments.

## UNIT OBJECTIVES

When you have completed this unit you will be able to:

1. Draw a diagram showing how mechanical switches can be connected to an MPU.
2. Explain how the MPU can eliminate the effects of contact bounce.
3. Explain the operation of a program that detects contact closure of switches, provides for debouncing, and decodes a simple keyboard.
4. Draw a simplified block diagram of a PIA and explain the purpose of the output, control, and data direction register.
5. Write a simple program that will configure the PIA in any desired input-output combination.
6. Explain how the PIA can be used to drive displays and encode keyboards.

## UNIT ACTIVITY GUIDE

- ☐ Play Cassette Tape Section “Designing with Microprocessors.”
- ☐ Read Section on Interfacing with Switches.
- ☐ Complete Self-Test Review Questions 1 through 11.
- ☐ Read Section on the Peripheral Interface Adapter (PIA).
- ☐ Complete Self-Test Review Questions 12 through 20.
- ☐ Read Section on Using the PIA.
- ☐ Complete Self-Test Review Questions 21 through 28.
- ☐ Perform Interfacing Experiments 5 through 9.
- ☐ Play Cassette Tape Section “Comparing Microprocessors.”
- ☐ Complete Unit Examination.
- ☐ Check Examination Answers.
- ☐ Complete Final Examination (optional).
- ☐ Mail Final Examination in the envelope provided (optional).

## INTERFACING WITH SWITCHES

The most popular input device used with the microprocessor is the switch. The operator of microprocessor-based equipment usually communicates with the MPU by using keyboard switches. However, in completely automated systems, the equipment being controlled often communicates with the MPU by limit switches, pressure switches, etc. In this section, you will examine some techniques used in interfacing with switches.

### Interfacing Requirements

When interfacing with a switch, or switches, four operations are involved. First, the MPU must select or address the proper switch bank. Second, it must detect contact closure. Third, it must provide for debouncing (unless this is accomplished by external hardware). Finally, it must decode the input. The following information describes each of these operations in more detail.

**Selecting the Switch** Figure 8-1 shows a simple arrangement for connecting eight switches to the MPU. Three-state buffers are used to interface the switches with the data bus. The buffers are enabled by the output of the address decoder. This decoder can use any of the decoding schemes discussed earlier. Assume that the decoder responds to address  $C003_{16}$ . Until the decoder receives this address, the buffers are disabled. This effectively disconnects the switches from the data bus.

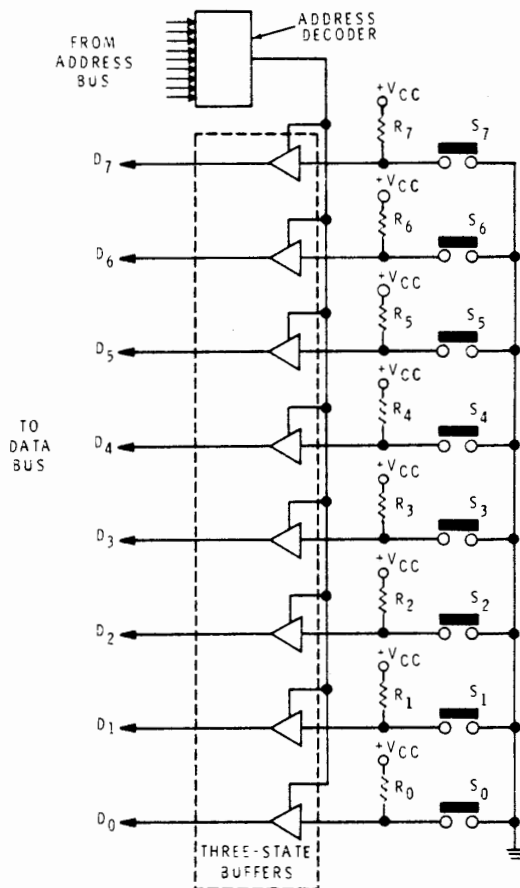
To find out if a switch is closed, the MPU must read in the data from this address. An easy way to do this is with the LDAA instruction. When the LDAA C003 instruction is executed, the address C003 goes out on the address bus. The decoder detects this address and enables the three-state buffer. Thus, for an instant, the switch bank is connected to the data bus. The data from the switch bank is loaded into accumulator A.

**Detecting Contact Closure** If none of the switches are closed, all the data lines will be high because of pull-up resistors  $R_0$  through  $R_7$ . Thus, the data entered into accumulator A will be  $FF_{16}$ . To test for a switch closure, the contents of accumulator A can be compared with  $FF_{16}$ . That is, a CMPA#FF instruction could be used. If this is followed by a BNE instruction, the MPU will branch if a key is depressed. Otherwise, it will not. For example, suppose  $S_0$  is closed. When the accumulator is loaded from address  $C003_{16}$ , the  $D_0$  line will be low. Thus, the number loaded into the accumulator will be  $FE_{16}$ . The CMPA instruction clears the zero flag since no match occurs. Therefore, the BNE instruction causes the branch to occur.

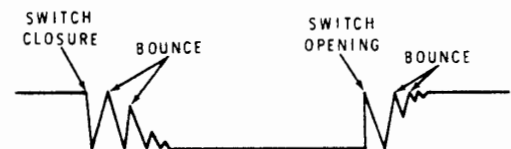
**Debouncing the Switch** Most mechanical switches produce contact bounce. When the switch is closed, the contacts do not make an immediately solid electrical or mechanical connection. Instead, they “bounce” open and closed for a brief period of time. Figure 8-2 illustrates this effect.

While contact bounce may only last for a few milliseconds, this is long enough for the MPU to interpret the bounce as repeated switch closures. To overcome this bounce problem, some switches use cross-coupled NAND gates that will immediately latch in one state so that all contact bounce is ignored. However, this requires additional circuitry.

In many applications, a better approach is to let the MPU itself do the debouncing. A simple scheme is to wait about ten milliseconds and then read in the data from the switch bank again. If the same indication occurs, then the MPU can be certain that the switch is closed. The switch can be checked as many times as is necessary to ensure that contact bounce is eliminated.



**Figure 8-1**  
Interfacing a bank of switches to the MPU.



**Figure 8-2**  
The effects of contact bounce.

**Decoding the Switches** After the MPU determines that a switch has been closed, it must decide which switch it is. In most cases the switch closure represents a number. For example, the MPU should recognize an  $S_5$  closure as the number 5. This, too, is easily accomplished by the proper subroutine.

Referring to Figure 8-1, you can see that each switch corresponds to one bit of the data line. When a switch is closed, the corresponding data line goes to 0. When loaded into the accumulator, the corresponding bit is also 0. The bit that is 0 can be detected by rotating the accumulator into the carry bit until the carry bit is cleared. By counting the number of rotations, the MPU can determine which switch is depressed.

In most applications, another job of the decoding procedure is to reject multiple switch closures. If two switches are closed simultaneously, the MPU should not accept data. If a second switch is closed before the first switch is released, the MPU may reject the data or accept only the first switch closure. By using a few extra programming steps, a very simple and inexpensive keyboard can appear quite sophisticated.

## A Typical Keyboard Arrangement

The keyboard arrangement used with the ET-3400 Microprocessor Trainer is a good example of what can be done with simple switches. A simplified circuit is shown in Figure 8-3.

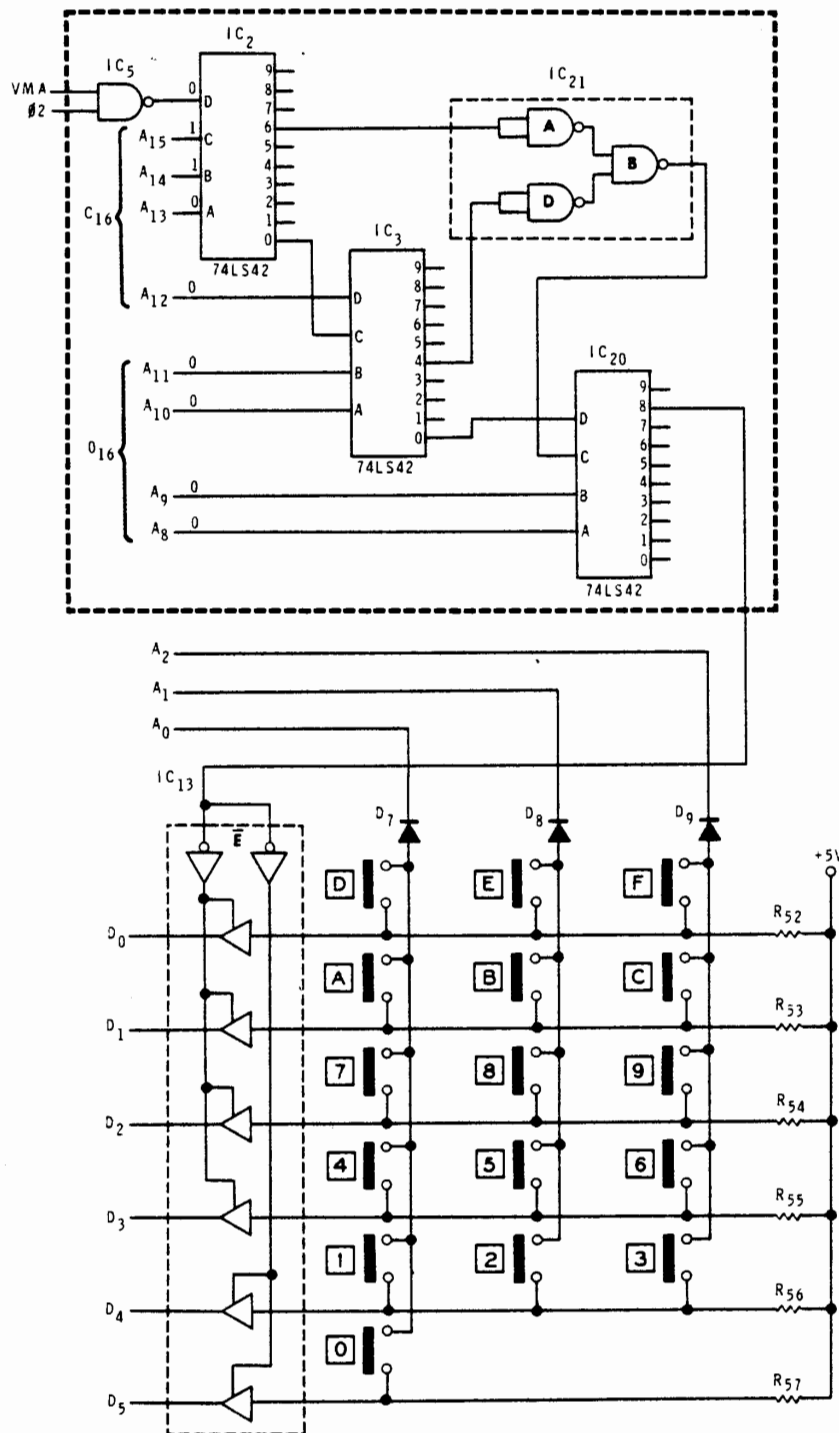


Figure 8-3  
Keyboard arrangement of the ET-3400  
Microprocessor Trainer.

**The Circuit** The address decoder is shown in dotted lines at the top of the Figure. For the most part it consists of three 74LS42 decoders. (The operation of this type of decoder was discussed in an earlier experiment.) The truth table for the 74LS42 is repeated in Figure 8-4.

The address decoder is also used to select the 7-segment displays. However, in this unit, we will be concerned only with keyboard decoding.

The portion of the address decoder shown in Figure 8-3 monitors address lines  $A_8$  through  $A_{15}$ . That is, it monitors the high order address. The keyboard is selected by enabling IC13. Normally, IC13 is in its high impedance state so that the keyboard is isolated from the data bus. IC13 is enabled by applying logic 0 to the enable line ( $\overline{E}$ ).

The address decoder enables IC13 when the high order address is  $CO_{16}$ . With a high order address of  $CO_{16}$ , address lines  $A_{15}$  and  $A_{14}$  are at logic 1 while  $A_8$  through  $A_{13}$  are at logic 0. Decoder IC2 is controlled by address lines  $A_{13}$ ,  $A_{14}$ ,  $A_{15}$ , and the VMA and  $\phi 2$  clock signal. When an instruction such as LDA C003 is executed, the inputs to IC2 will be as shown in Figure 8-3. The truth table for the decoder shows that output line 6 will be logic 0 while all other output lines will be logic 1.

DEC. BCD INPUT					OUTPUT LINES									
NO.	D	C	B	A	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	1	1	1	1	1	1	1	1	1
1	0	0	0	1	1	0	1	1	1	1	1	1	1	1
2	0	0	1	0	1	1	0	1	1	1	1	1	1	1
3	0	0	1	1	1	1	1	0	1	1	1	1	1	1
4	0	1	0	0	1	1	1	1	0	1	1	1	1	1
5	0	1	0	1	1	1	1	1	1	0	1	1	1	1
6	0	1	1	0	1	1	1	1	1	1	0	1	1	1
7	0	1	1	1	1	1	1	1	1	1	1	0	1	1
8	1	0	0	0	1	1	1	1	1	1	1	1	0	1
9	1	0	0	1	1	1	1	1	1	1	1	1	1	0
>9	INVALID CODES				1	1	1	1	1	1	1	1	1	1

Figure 8-4  
Truth table for the 74LS42 decoder.



The 1 at output 0 of IC2 is applied to IC3. Address lines  $A_{10}$  through  $A_{12}$  are also connected to IC3. With a high order address of  $CO$ , these lines will be at 0. With these inputs, the truth table shows that output line 4 of IC3 will be 0 while all other outputs will be 1. The 0 at output line 4 is inverted by IC21D. Simultaneously, the 0 at output 6 of IC2 is inverted by IC21A. These two signals are then NANDed together to form logic 0 at the output of IC21B. This 0 is applied to input C of IC20.

The other inputs to IC20 include a logic 1 from IC3, logic 0 from  $A_9$ , and logic 0 from  $A_8$ . The truth table shows that this will result in a logic 0 at output 8 of IC20. This logic 0 is applied to the enable input of IC13. This enables the three-state buffers and momentarily connects the keyboard to the data bus. Thus, the keyboard is momentarily connected to the data bus any time the high order address is  $CO_{16}$ .

The keyboard is divided into three columns. The center and right columns have five keys each while the left column has six keys. The RESET key is not shown since it is not addressed as the other keys are. The low order address determines which column of keys is selected.  $A_0$  controls the left column and  $A_1$  and  $A_2$  control the center and right columns, respectively. A column is selected by choosing an address that will force the desired column line low, but will hold the unwanted column lines high. Address  $C003_{16}$  fulfills these requirements for the right-hand column of keys. So do many other addresses, but consider this to be the address of that column. In the same way, the center column has an address of  $C005_{16}$  and the left column has an address of  $C006_{16}$ .

**Detecting and Encoding a Key Closure** The monitor program in the ROM of the Trainer has a subroutine called ENCODE which starts at address  $FDBB_{16}$ . The purpose of this subroutine is to look over the keyboard, determine if a key has been depressed, and produce the proper hexadecimal value of the depressed key. The hexadecimal value is placed in accumulator A. Also, the carry flag will indicate whether or not a valid key entry has occurred. A valid entry is defined as one and only one key depressed. The C flag will be set if the entry was valid. It will be cleared for nonvalid entries or no entries at all.

The ENCODE subroutine is shown in Figure 8-5. The following explanation will refer to the instructions by the line numbers given on the left. Notice that the subroutine is written in assembly language.

The first instruction saves the original contents of accumulator B. As you will see later, the program normally comes here from another subroutine called INCH. When it does, B will hold a timing count that must not be lost.

LINE	ASSEMBLY CODE		COMMENTS
1	ENCODE	PSH B	Save contents of accumulator B.
2		LDA B COL1	Load the right column into B.
3		LDA A COL3	Load the left column into A.
4		ASL A	
5		ASL A	Get rid of "don't care" bits.
6		ASL A	
7		ROL B	Double precision shift left
8		ASL A	of accumulators A and B
9		ROL B	to get rid
10		ASL A	of "don't care" bits.
11		ROL B	
12		PSH B	Save contents of B.
13		LDA B COL2	Load the center column into B.
14		AND B #\$1F	Mask off bits 5, 6, and 7.
15		ABA	Merge with A.
16		PUL B	Restore B.
17		COM A	After complementing the keyboard
18		COM B	pattern will be in A and B (1 = key closed).
19		STX T0	Save contents of index register.
20		LDX #HEXTAB-1	Point index register to table of hex values.
21		CBA	Which accumulator contains a 1?
22		BEQ ENC3	Neither or both contain 1's (invalid entry).
23		BCC ENC1	A contains a 1 so go to ENC1.
24		PSH A	B contains a 1 so
25		TBA	Swap the contents
26		PUL B	of A and B.
27		LDX #HEXTAB+7	Point the index register to upper half of hex table.
28	ENC1	TST B	None of these keys should be closed.
29		BNE ENC3	If they are, go to ENC3.
30	ENC2	INX	Otherwise, have the index register
31		ASLA	Scan up the table until it
32		BHI ENC2	finds the proper hex value.
33		BEQ ENC4	If only one key is depressed, entry is valid.
34	ENC3	CLC	Entry is not valid so clear C.
35	ENC4	LDA A O,X	Load the hex value into accumulator A.
36		LDX T0	Restore index register and
37		PUL B	accumulator B to their original values.
38		RTS	Return

Figure 8-5  
ENCODE subroutine.

The next two instructions load A and B from the keyboard columns. After line 3, A and B will contain the data from the two outside keyboard columns. Figure 8-6A shows which keys are associated with which bits. The indicated bit will contain 0 if its associated key is depressed. Otherwise, it will contain a 1. The X's are shown in those bits that are not affected by the keys. The first step is to eliminate these "don't care" states. In accumulator A and the carry flag, this is done by shifting to the left. After line 6, the accumulators and carry flag will contain the keyboard patterns shown in Figure 8-6B.

Next A and B are shifted left together through the carry flag. Figure 8-6C shows the contents of these registers after line 11. The contents of B are then saved by pushing them into the stack. Accumulator B is now free to be loaded with the data from the center column of the keyboard. Bits 5, 6, and 7 are masked off leaving the registers as shown in Figure 8-6D. B is added to A so that the 0's in A are replaced with the states of keys 2, 5, 8, B, and E. When B is pulled from the stack (line 16), the registers will contain the keyboard pattern as shown in Figure 8-6E. Notice that each of the 16<sub>10</sub> keys is represented by one of the bits in the accumulators. If no keys are depressed, all bits will be 1's. If a key is depressed, its corresponding bit will be 0.

In lines 17 and 18, the two accumulators are complemented. Thus, from this point on a depressed key is represented by a 1 while an open key is represented by a 0.

Assume that the D key is depressed. In this case, the accumulators will appear as shown in Figure 8-6F after line 18. In line 19, the contents of the index register are saved in a temporary location in RAM called TO. Next the index register is loaded with one less than the starting address of a hexadecimal table.

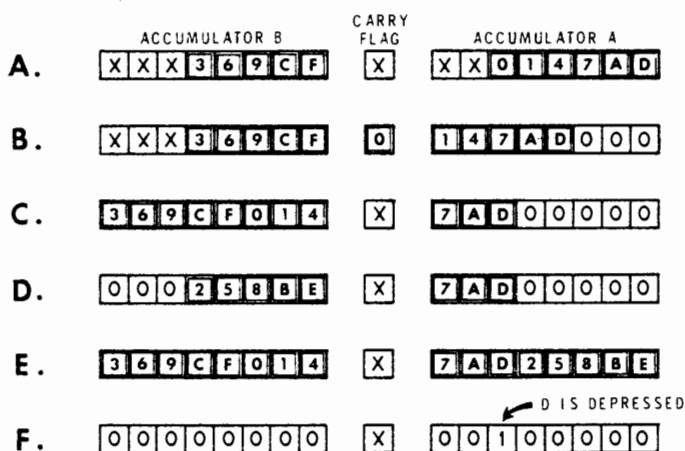


Figure 8-6  
The keyboard bit pattern is placed in accumulators A and B.

The hexadecimal table is shown in Figure 8-7. Notice that the hex digits 00 through 0F are not in order in the table. However, compare the entries in the table to Figure 8-6E. The first eight entries in the table are in the same order as the keyboard pattern in accumulator A. The upper eight entries correspond to the key patterns in accumulator B. As you will see later, this is no accident.

Line 21 of the program compares the contents of accumulators A and B. With D depressed, the number in A will be larger. The result is that both the Z and C flags are cleared. The BEQ does not cause a branch because the Z flag is cleared. However, the BCC instruction does cause a branch because the C flag is cleared. Notice that the branch is to the point labelled ENC1 (line 28).

<u>HEX ADDRESS</u>	<u>HEX CONTENTS</u>	<u>SYMBOLIC ADDRESS</u>
FFA6	07	HEXTAB
FFA7	0A	
FFA8	0D	
FFA9	02	
FFAA	05	
FFAB	08	
FFAC	0B	
FFAD	0E	
FFAE	03	
FFAF	06	
FFB0	09	
FFB1	0C	
FFB2	0F	
FFB3	00	
FFB4	01	
FFB5	04	

Figure 8-7  
The hexadecimal table (HEXTAB).

This part of the subroutine encodes the key closure and at the same time checks to see that no other keys are closed. The first step tests B. If the result is not zero, the MPU knows that a second key is closed and the entry is ignored by branching to ENC3. (The result of this will be shown later). Otherwise, the index register is incremented to the address of the first entry in the hex table, the contents of accumulator A is then shifted to the left, and the BHI instruction simultaneously checks both the C and Z flags. The branch is implemented if both C and Z are cleared. Both are cleared in this case so the program jumps back to ENC2 and the index register is incremented so that it points to the second entry in the table. Accumulator A is shifted left again. This places the 1 (that represents switch D being closed) into the MSB of the accumulator. C and Z are still cleared so the BHI instruction sends the program back to ENC2 again.

The index register is incremented again so that it now points to the third entry in the hex table. Notice that the third entry is  $OD_{16}$ . Thus, the index register is now pointing to the number that corresponds to the switch closure. Accumulator A is shifted left again so that the 1 (representing switch D) is placed in the carry flag. This sets the C flag and, consequently, the BHI instruction cannot cause a branch.

Because the BHI branch does not occur, the next instruction encountered is the BEQ instruction. If only one key is depressed, the contents of accumulator A should be zero. If it is zero, then only one key was depressed. Otherwise, a second key was depressed and the entry should be tagged not valid. If the entry is valid, the BEQ instruction causes the program to jump over the CLC instruction to ENC4 (line 35).

At ENC4, accumulator A is loaded with the third entry from the hex table. Thus, the program has fulfilled its requirements. A number corresponding to the key depressed is in accumulator A and the C flag is set indicating a valid entry. All that remains is to restore the original contents of the index register and accumulator B. Finally, the RTS instruction returns the program to the point where this subroutine was called.

If you step through this subroutine with a different key depressed, you will see that the proper hex code is always returned. If no keys are depressed or if two keys are depressed, the program will branch to ENC3. This clears the carry flag. Thus, the subroutine will end with C cleared, which indicates that the entry was not valid.

**Eliminating Contact Bounce** A second subroutine is used to eliminate contact bounce. It is called INCH for "input character." Its starting address is FDF4 and it calls the ENCODE program just discussed  $20_{16}$  different times. If for  $20_{16}$  consecutive times, ENCODE tells INCH that a valid entry exists, INCH is convinced and accepts the entry. Since this process requires several milliseconds, any contact bounce is eliminated.

This subroutine is shown in Figure 8-8. The subroutine is divided into two nearly identical halves. The first half (lines 1 through 6) waits for the keyboard to be cleared (no keys depressed). It does this so that it does not mistake a previous key closure for a valid entry.

If a key is depressed upon entering this subroutine, first, the contents of B are saved; then B is loaded with a delaying count of  $20_{16}$ . By counting this number down to zero, the program establishes a delay sufficient to "wait out" any contact bounce.

LINE	ASSEMBLY CODE			COMMENTS
1	INCH	PSH B		Save contents of accumulator B
2	INC1	LDA B	#\$20	Load B with a delaying count of $20_{16}$ .
3	INC2	BSR	ENCODE	Branch to the ENCODE subroutine.
4		BCS	INC1	If carry is set go to INC1.
5		DEC B		Otherwise decrement the count.
6		BNE	INC2	If count is not zero go back to INC2.
7	INC3	LDA B	#\$20	Load B with a delaying count of $20_{16}$ .
8	INC4	BSR	ENCODE	Branch to the ENCODE subroutine.
9		BCC	INC3	If carry is clear, check again.
10		DEC B		Otherwise, decrement the count.
11		BNE	INC4	If count is not zero go back to INC4
12		PUL B		Restore the original contents of accumulator B.
13		RTS		Return.

Figure 8-8  
INCH subroutine.

The BSR instruction sends the MPU off to the ENCODE subroutine. If a key is depressed, the C flag will be set when the program returns. The BCS instruction sends the program back to INC1 if the C flag is set. The program will stay in this loop until the ENCODE subroutine returns with the carry bit clear. Of course, this will happen only after a key is released. This prevents a single entry from being mistaken for two or more entries. An entry is accepted only after the previous entry is released.

Once a key is released, ENCODE will clear the carry flag. Thus, the BCS instruction will not cause a branch. Instead B is decremented and checked for zero. If not zero, the program branches back to INC2. This loop is repeated  $20_{16}$  times and gets rid of any contact bounce associated with the release of the previous key. Once the program is convinced that the previous key has been released, it proceeds to the second half of the subroutine (lines 7 through 13).

Accumulator B is loaded with a delaying count of  $20_{16}$  again and the BSR instruction calls the ENCODE subroutine. Upon return from this subroutine, the BCC instruction checks the carry flag. If it is clear, no valid entry is being received and the program branches back to INC3. In practice, the MPU in the ET-3400 Trainer spends most of its time caught in this loop waiting for a key closure to occur.

When a key closure does occur, the ENCODE subroutine sets the carry bit. This allows the MPU to escape the loop. It then enters the loop composed of lines 8 through 11. It repeats this loop  $20_{16}$  times to eliminate any contact bounce. When it escapes this loop, we can be confident that the key closure is absolutely valid.

This is a good example of software-hardware trade offs. These subroutines make the keyboard appear quite sophisticated. A mechanical or electromechanical keyboard having all these features would be very expensive.

You will learn more about interfacing with switches later after you learn about the peripheral interface adapter. Also, in a later experiment you will gain some practical experience interfacing switches.

## Self-Test Review

1. List four requirements that must be met when connecting mechanical switches to the MPU.
2. What type of circuit is often used between the switches and the data bus of the MPU?
3. Refer to Figure 8-1. If no switches are closed, what hexadecimal number is read from the switch bank?
4. Refer to Figure 8-1. If  $S_2$  is closed, what hexadecimal number is read from the switch bank?
5. What two things can be determined by the hexadecimal number read in from the switch bank?
6. Why is debouncing required?
7. How can the MPU overcome the effects of contact bounce?
8. Refer to Figure 8-3. To what address does the center row of keys respond?
9. Refer to Figure 8-5. What is the purpose of the first 18 lines of the program?
10. What technique is used for finding the hexadecimal equivalent of the key that is depressed?
11. Refer to Figure 8-8. How does this routine overcome contact bounce?



**ANSWERS**

1. The MPU must:
  - 1) Address the switches.
  - 2) Detect contact closure.
  - 3) Overcome contact bounce.
  - 4) Decode the switches.
2. A three-state buffer.
3.  $FF_{16}$ .
4.  $FB_{16}$ .
5. The hexadecimal number read from the switch bank reveals
  - 1) If a switch is closed.
  - 2) Which switch is closed.
6. When a mechanical switch is closed, its contacts often bounce open and closed several times. Unless this is taken into consideration, a single switch closure can be mistaken for multiple switch closures.
7. The MPU can overcome the effects of contact bounce by rechecking the closed switch several times.
8.  $C005_{16}$ .
9. The first 18 lines of the program place the switch pattern into accumulators A and B.
10. They are looked up in a table.
11. By rechecking the closed switch. It must find the switch closed for  $20_{16}$  consecutive checks before the value is accepted as good.

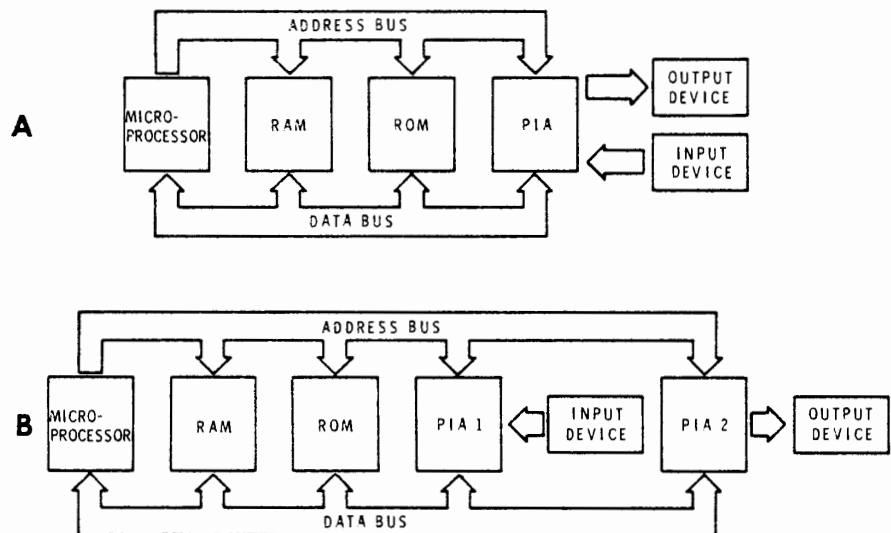
## THE PERIPHERAL INTERFACE ADAPTER (PIA)

Most microprocessors have a family of support chips that are used to simplify the problem of interfacing with the outside world. One of the most popular of these interfacing chips is the 6820 peripheral interface adapter (PIA). The PIA was developed to support the 6800 MPU. However, it is also being used in many microprocessor based designs using other MPU's.

While a complete discussion of support chips is beyond the scope of this course, detailed data sheets on several support IC's are included in Appendix B.

In this section, you will be introduced to the PIA. You will learn enough about it to use it in the experiments that follow.

The block diagrams of two typical systems that use PIA's are shown in Figure 8-9. In Figure 8-9A a single PIA is used to drive both an input and an output device. This is possible since the PIA has two independent channels. Figure 8-9B shows a system that uses two PIA's. One controls an input device while the other controls an output device.



**Figure 8-9**  
The PIA is used to interface input and output devices to the MPU.

The purpose of the PIA is to simplify the problem of interfacing the MPU to external devices. Of course, any device can be interfaced with the MPU using conventional combinational logic. However, the conventional logic approach generally requires many IC's. This defeats one of the prime advantages of the microprocessor — a simple straightforward design requiring few IC's. The advantage of the PIA is that in many cases one or two IC's can do all the interfacing.

Because the PIA can do most routine peripheral control tasks, the MPU is freed to handle more important tasks. Also, the PIA allows the MPU to treat a peripheral device as a memory location. In addition, it acts as a buffer between the high-speed MPU and the low-speed I/O device. Since the PIA has some on board address decoding, a separate address decoder is not needed in many applications.

The PIA is superior to combinational logic in another way. The PIA is extremely flexible because it is programmable. That is, its configuration can be changed from one moment to the next by the program being executed. For example, an output port can be changed to an input port in the middle of a program. Later, you will see how this is done. But first, you will learn about the internal structure of the PIA.

## I/O Diagram

The diagram of the PIA shown in Figure 8-10 shows its input and output lines. Since this is an interface device, one side connects to the MPU while the other side connects to one or more peripheral devices.

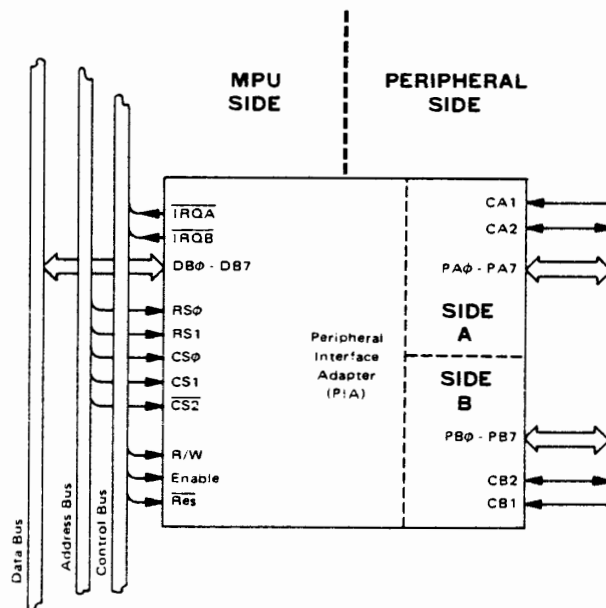


Figure 8-10  
I/O diagram of the peripheral interface  
adapter.

On the MPU side, the PIA monitors several address, data, and control lines of the MPU. It monitors all eight data bus lines. Data is transferred to and from the PIA, a byte at a time, by the data bus. For this reason, the PIA is said to be "byte oriented." In this respect, the PIA is treated like memory. As you will see later, the PIA has four addresses that can be treated much like RAM.

The PIA can monitor five of the MPU's address lines. This is enough to partially decode the address. In many cases, no additional address decoding is necessary.

The PIA also connects to several control lines. The  $R/\overline{W}$  line informs the PIA whether it is to receive data from the MPU or send data to the MPU. Once again, note the similarity between the PIA and RAM.

Another similarity is the enable line. Like RAM, the PIA is enabled by the  $\phi 2$  clock (often ANDed with VMA). This provides the basic timing signal for the PIA.

The reset line of the PIA is generally connected to the master system reset. This allows the PIA registers to be reset to a known condition at the same time the MPU is reset.

The PIA has two interrupt request lines. These allow the PIA to request service from the MPU. These may connect to either the IRQ or the NMI lines of the MPU. As you saw earlier, interrupts can be used to simplify I/O operations and to save MPU time. While the interrupt capability of the PIA is not discussed in this unit, the PIA data sheet in Appendix B briefly outlines these capabilities.

The peripheral side of the MPU has two nearly identical I/O channels.  $PA_0$  through  $PA_7$  make up a peripheral data bus for the A side of the PIA.  $CA_1$  and  $CA_2$  are two control/interface lines associated with the A side. Notice that the B side has comparable data and control lines. Do not confuse the peripheral data buses on the right with the MPU data bus on the left. Both buses will be referred to frequently in the following discussion.

## PIA Registers

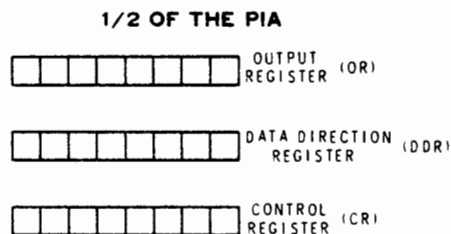
The A and B sides of the PIA are nearly identical. Except when noted, everything stated about one side of the PIA also applies to the other side.

Each side of the PIA has three main registers as shown in Figure 8-11. These include an output register (OR), a data direction register (DDR), and a control register (CR). The output register is used to hold a data byte that is being transferred to the peripheral data bus. It acts as a temporary storage location for data being transferred from the MPU to the peripheral device.

The data direction register sets up the individual lines in the peripheral data bus as either inputs or outputs. Each bit in the DDR controls the corresponding peripheral data line. A 1 in a specific bit of the DDR causes the corresponding peripheral data line to act as an output line. A 0 causes it to act as an input line.

To set up all eight peripheral data lines of the A side as inputs, we simply store  $00_{16}$  in the DDR of the A side. By the same token, we can set up the B side as output data lines by storing  $FF_{16}$  in the DDR of the B side. The various data lines can be set up in any combination. Moreover, a peripheral data line can be changed from input to output simply by changing its corresponding bit in the data direction register. Keep in mind, that this change is made by software. No hardware change is necessary.

The control register allows you to program several other characteristics of the PIA. One of these will be discussed later. The others are explained in the PIA data sheets in Appendix B.



**Figure 8-11**  
Each side of the PIA has three main registers.

### Addressing the PIA Registers

The PIA has six registers in which data can be stored and from which data can be read. The two control registers each have an address of their own. In a typical system, the control register on the A side may have an address of  $4005_{16}$ . The control register on the B side may have an address of  $4007_{16}$ . In this case, we could write data into the control register on the A side with the instruction `STAA  $4005_{16}$` . Or, we could read from the control register on the B side with `LDAA  $4007_{16}$` .

While the control register has an address of its own, the data direction register and the output register share a common address. Typically, the data direction and output register on the A side may share the address  $4004_{16}$ . Those on the B side may share the address  $4006_{16}$ . Thus, in a typical system, the PIA may have addresses assigned as shown in Figure 8-12.

Even though the data direction and output registers share the same address, each is still individually accessible. When address 4004 appears on the address bus, either ORA or DDRA will be selected depending on bit 2 of the control register. If bit 2 of CRA is a 1, then address  $4004_{16}$  selects ORA. However, if bit 2 of CRA is 0, then address 4004 selects DDRA. In this same way, bit 2 of CRB determines which register is selected by address  $4006_{16}$ . A 1 selects the output register; a 0 selects the data direction register.

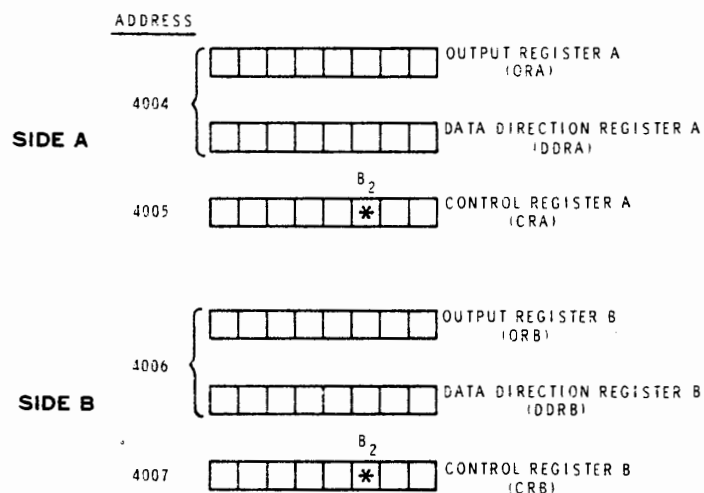
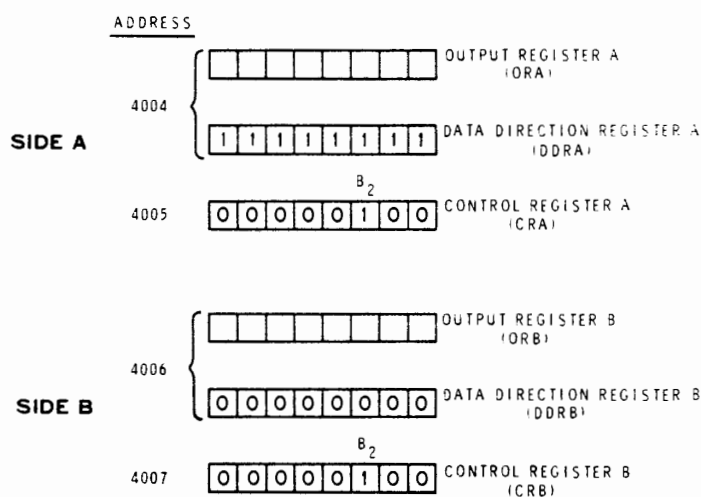


Figure 8-12  
Typical address assignments of PIA  
registers.

## Initializing the PIA

Before the PIA can be used for input-output transfers, it must first be programmed to operate in the desired manner. For example, assume that you wish to use the A side of the PIA as an output port and the B side as an input port. Figure 8-13 shows a PIA that is configured in this manner. DDRA sets all A-side peripheral data lines as outputs and bit 2 of CRA has set the output register to respond to address 4004. DDRB sets all B-side peripheral data lines as inputs and bit 2 of CRB has set ORB to respond to address 4006. This state does not come about by accident. We must deliberately set up these conditions. This process is called initializing the PIA.

In most applications, the PIA is initialized after a system is reset. Once configured in a certain way, the PIA is normally left in this configuration. For this reason, you can assume that the initialization process starts immediately after the system is reset.



**Figure 8-13**  
The A side is configured as an output  
port; the B side, as an input port.

The PIA has a reset line that is normally connected to the system reset line. When the PIAs' reset line goes low, all the registers in the PIA are reset to zero as shown in Figure 8-14. With both data direction registers reset to zero, both peripheral data buses are configured as inputs. Also, with bit 2 of the control registers reset to 0, address 4004 selects DDRA while 4006 selects DDRB. To initialize the PIA, we must change the contents of its registers from that shown in Figure 8-14 to that shown in Figure 8-13.

A program that will accomplish this is:

```
LDAA    #FF
STAA    4004
LDAA    #04
STAA    4005
STAA    4007
```

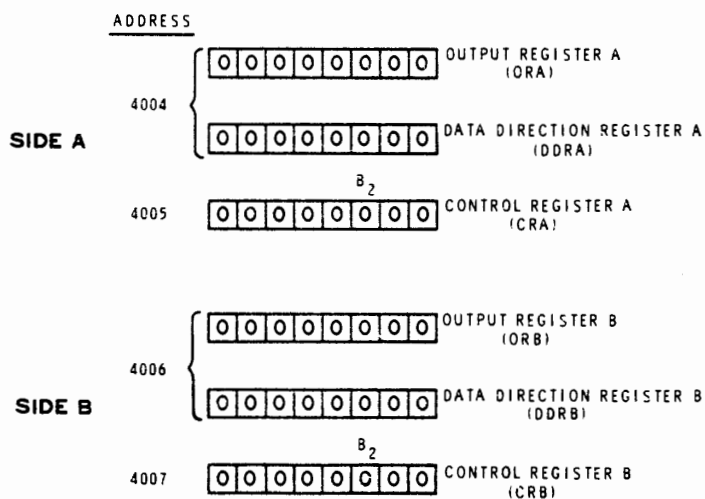


Figure 8-14

When the PIA is reset, all registers are set to zero.

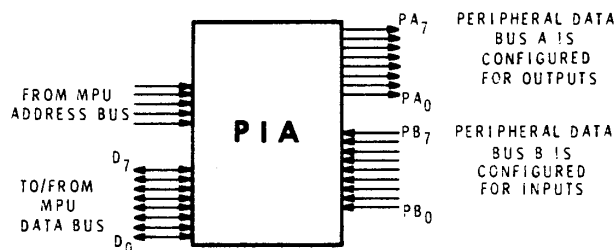


The first instruction loads accumulator A with all binary 1's. The second instruction stores this at location  $4004_{16}$ . Since bit 2 of CRA is initially 0, FF is stored in DDRA. This configures the A-side peripheral data bus as outputs.

The third instruction loads  $04_{16}$  into accumulator A. The next two instructions store  $04_{16}$  at addresses  $4005_{16}$  and  $4007_{16}$ . These addresses are the control registers. Recall that  $04_{16}$  is equal to  $0000\ 0100_2$ . This sets bit 2 of both control registers to 1. Consequently, address  $4004_{16}$  now specifies ORA while  $4006_{16}$  now specifies ORB.

As you can see, the PIA is now set up as shown in Figure 8-15. Notice that we did not have to change the contents of DDRB in this case because it was initially reset to zero.

Once the PIA is configured in this manner, the MPU can transfer data to the output port using the instruction: STAA  $4004_{16}$ . Also, it can read data from the input port using the instruction: LDAA  $4006_{16}$ .



**Figure 8-15**  
The initialization procedure configures the PIA as shown.

## Addressing the PIA

Figure 8-16 shows how the PIA fits into a microprocessor system. Now consider how the PIA is addressed.

The PIA has three chip select lines (CS0, CS1, and  $\overline{\text{CS2}}$ ). These three lines are used to select the PIA. In order for this particular PIA to be selected, CS0 and CS1 must be high while  $\overline{\text{CS2}}$  must be low.

Notice that these three lines are connected to three of the address lines (A2, A14, and A15). In this application, the A14 line is ANDed with the VMA line. This ensures that the PIA is selected only if the address is valid. In the following discussion, assume that all addresses are valid.

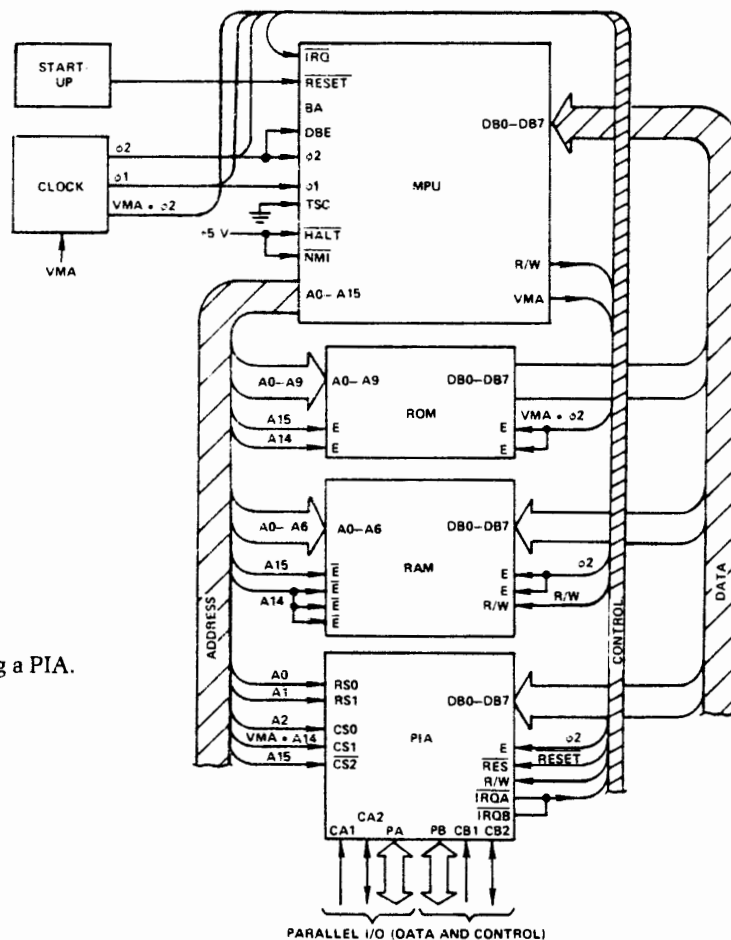


Figure 8-16  
A microprocessor system using a PIA.

The PIA will be selected by any address in which A2 and A14 are high and A15 is low. Hexadecimal addresses like 4004, 5004, 6004, 7004, etc. will select the PIA because the above conditions are met. Actually, there are thousands of addresses that will select the PIA. Even so, in many systems this is no problem. In the application shown, any address below  $3FFF_{16}$  will select the RAM. Many addresses between  $4004_{16}$  and  $7FFF_{16}$  select the PIA. Finally addresses above  $C000_{16}$  select the ROM. Neither the ROM, the RAM, nor the PIA is fully decoded. Even so, their addresses are unique enough that each can be selected without additional decoding.

For programming purposes, we must assign the PIA four consecutive addresses. We will assume that these addresses are  $4004_{16}$  through  $4007_{16}$ . Notice that we could have just as easily selected addresses  $6004_{16}$  through  $6007_{16}$ .

In addition to the chip select lines, the PIA has two register select lines (RS0 and RS1) that also connect to the address bus. RS0 connects to A0 while RS1 connects to A1. The RS1 line determines which side of the PIA is selected. When RS1 is at logic 0, side A is selected. When RS1 is at logic 1, side B is selected.

The RS0 line selects the register on the affected side. When RS0 is 1, the control register is selected. When RS0 is 0, the data direction register or the output register is selected depending on the state of bit 2 of the control register.

HEX ADDRESS	BINARY EQUIVALENT OF LAST HEX DIGIT			REGISTER SELECTED
	CS0	RS1	RS0	
4004	0	1	0	DDRA or ORA*
4005	0	1	0	CRA
4006	0	1	1	DDRB or ORB*
4007	0	1	1	CRB

\*Determined by bit 2 of CR  
0 = DDR  
1 = ORA

Figure 8-17  
The relationship between the address  
and the register selected.

The chart shown in Figure 8-17 shows why the various registers are selected for the addresses shown. For example, when the address is  $4007_{16}$ , address lines 1 and 0 are both high. Thus, both RS0 and RS1 are at logic 1. The 1 at RS1 selects the B side of the PIA. The 1 at RS0 selects the control register. Thus, the address  $4007_{16}$  selects the control register on the B side. Figure 8-18 illustrates the same thing in another way. It shows how the data path between the MPU and the PIA register is determined.

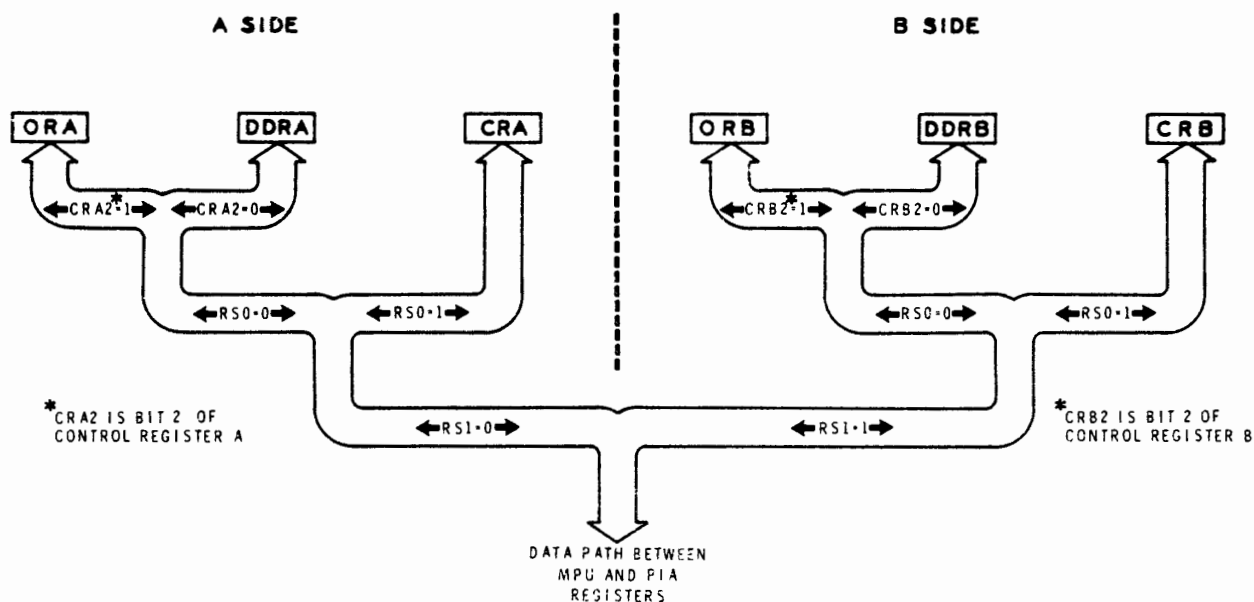


Figure 8-18  
How the data path between the MPU  
and the PIA register is determined.

## Self-Test Review

12. What is the peripheral interface adapter (PIA)?
13. How is the PIA superior to combinational logic?
14. What is the internal structure of the PIA?
15. How is the PIA reset?
16. What are the contents of the PIA registers immediately after being reset?
17. How does the MPU decide which side of the PIA is selected?
18. Once the MPU has selected one side of the PIA, how does it determine which of the three registers is connected to the data bus?
19. Which pin of the PIA is normally connected to the A1 address line of the MPU?
20. Refer to Figure 8-16. Write a short routine that will initialize the PIA immediately after reset. Set up PA0 through PA3 and PB0 through PB3 as inputs. Set up PA4 through PA7 and PB4 through PB7 as outputs.

**ANSWERS**

12. The PIA is a 40-pin IC that is used to simplify the transfer of data between a microprocessor and the outside world.
13. In many cases, one or two PIAs can handle all interfacing requirements. Also, the PIA is extremely flexible since it can be programmed to perform in several different configurations.
14. The PIA has two nearly identical sides, each of which contains three registers: the output register, the data direction register, and the control register. The MPU can transfer data to or from either of these registers by way of the data bus.
15. The PIA is reset by pulling its  $\overline{\text{reset}}$  line low.
16. When reset, the contents of all PIA registers are reset to  $00_{16}$ .
17. The MPU selects the A side of the PIA by switching the PIA's RS1 line to 0 (low). The B side is selected by switching the RS1 line to 1.
18. If the RS0 line of the PIA is 1, the control register of the affected side is selected. If RS0 is 0, then the register selection is determined by bit 2 of the affected control register.
19. RS1 of the PIA is normally connected to address line A1 of the MPU.
20. A typical routine is:

LDAA	#F0
STAA	4004
STAA	4006
LDAA	#04
STAA	4005
STAA	4007

## USING THE PIA

Now that you are familiar with the PIA, you are ready to examine some of the ways that the PIA can be used. In this section, you will see how the PIA can be used to handle displays and keyboards. You will start by examining how a PIA can be used to drive 7-segment displays.

### Driving 7-Segment Displays

Figure 8-19 shows how a single PIA can be used to multiplex up to eight 7-segment displays. The PIA is configured to respond to addresses  $4004_{16}$  through  $4007_{16}$ . The A side of the PIA is used to supply the 7-segment code to the displays. Inverters are used to supply the current required by the displays. The B side is used to determine which display is selected. Here discrete transistors are used to provide the required current.

The displays are common cathode types. To light segment “a” of display 1,  $Q_1$  must conduct through pin “a.” Thus, a logic 1 must be applied to the base of  $Q_1$  and to pin “a” of display 1.

Notice that both sides of the PIA must serve as outputs. Thus, during the initialization procedure, both data direction registers are set to  $FF_{16}$ . Then bit 2 of both control registers are set to 1’s, so that data would be routed to the output registers.

All displays are blanked by storing  $FF_{16}$  in output register A. This sets PA0 through PA7 to 1’s. The 1’s are then inverted to 0’s. Thus, no segments of any display can light.

To display a specific eight-character message, the displays must be turned on one at a time in sequence. This is controlled by the B side of the PIA. At the same time, the 7-segment character codes must be loaded into the A side of the PIA one at a time.

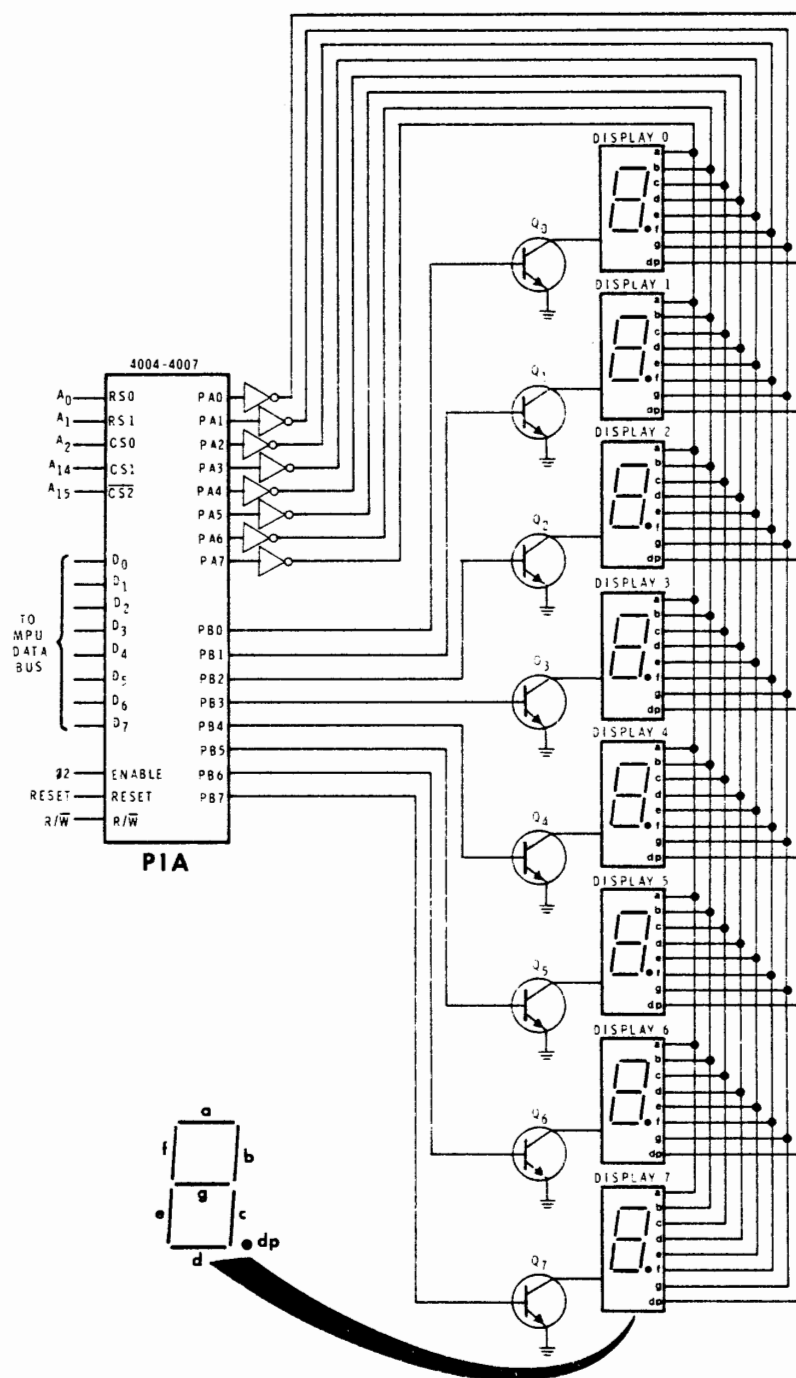


Figure 8-19  
Using the PIA to multiplex displays.



A subroutine for multiplexing the displays is shown in Figure 8-20. It assumes that the eight 7-segment codes are already in RAM at consecutive addresses, SGCODE through SGCODE + 7.

The first instruction loads the index register with one less than the address of the 7-segment code for the first character. Next, accumulator B is cleared and the carry flag is set to 1. Accumulator A is set to  $FF_{16}$  by first clearing to  $00_{16}$  and then complementing. The  $FF_{16}$  in accumulator A is stored in the output register of the A side of the PIA. This sets PA0 through PA7 to 1's. The ones are inverted and blank all the displays. The displays are blanked by side A whenever the display pointer (side B) is being changed.

The next instructions rotate accumulator B to the left through the carry flag. Recall that accumulator B was originally cleared and that the carry flag was set. Thus, the 1 in the carry flag is rotated into bit 0 of accumulator B. The new contents of accumulator B are stored in the B side of the PIA. PB0 is set to 1 while PB1 through PB7 are reset to 0. The 1 at PB0 enables display 0. However, the display still does not light because the segment lines are still at logic 0.

LINE	ASSEMBLY CODE			COMMENTS
1	DISPLAY	LDX	#SGCODE-1	Point to first code minus one.
2		CLRB		Initialize the
3		SEC		display pointer
4	NXDIGIT	CLRA		Set ACCA
5		COMA		to FF
6		STAA	PIAORA	Turn off all displays
7		ROLB		Point to next display in sequence.
8		STAB	PIAORB	Enable next display in sequence.
9		BCC	NXTDSP	If last display has been lit,
10		RTS		exit. Otherwise,
11	LTDSP	INX		point to next 7-segment code
12		LDAA	O,X	Load code into ACCA.
13		STAA	PIAORA	Display the code.
14		CLRA		Leave this
15	DELAY	INCA		display lit for
16		BNE	DELAY	1536 <sub>10</sub> MPU cycles.
17		BRA	NXDIGIT	Go back and do it again.

Figure 8-20  
Subroutine for multiplexing the displays.

The next instruction (BCC) checks the carry flag to see if it is cleared. It will be in this case because a 0 was rotated into it by the earlier ROLB instruction. Consequently, the RTS instruction is skipped over and the INX instruction is executed next.

The INX instruction increments the contents of the index register so that it now points at the 7-segment code for the first character that is to be displayed. The next instruction loads this character into accumulator A. Then the 7-segment code is stored in output register A of the PIA. The code is inverted and is applied to the segment lines of all eight displays. However, only display 0 is presently enabled. Thus, this is the only display that will be lit.

The next three instructions cause a delay of about  $1536_{10}$  MPU cycles. In a typical system, this amounts to a delay of about 2 milliseconds. Finally, the BRA instruction branches the program back to the point called NXDIGIT.

At NXDIGIT, the displays are blanked again. Accumulator B is rotated to the left so that the 1 now appears at bit 1. This is stored at PIAORB, enabling display 1 and disabling all other displays. The carry flag is still cleared, so the RTS instruction is skipped. The next 7-segment code is selected and stored at PIAORA. Thus, the second code lights display 1. The display is held lit for about 2 milliseconds and the loop is repeated again.

The display loop continues until all eight displays have been lit. After the final display is lit, the program tries to repeat the loop again. However, this time, the 1 in accumulator B is rotated back into the carry flag. As a result, the BCC instruction does not cause a branch. The RTS instruction is executed and the program returns to wherever it came from.

In order to give the illusion of a constant display, this subroutine must be called several times each second. The display must be constantly refreshed by rewriting the same message over and over again.

## Decoding Keyboards

Figure 8-21 illustrates how the PIA can be used to decode a 16-switch keyboard. The chip select lines are connected so that this PIA responds to addresses  $4008_{16}$  through  $400B_{16}$ .

One switch is connected to each of the peripheral data lines of the PIA. When a switch is open, its corresponding peripheral data line is pulled up to logic 1 by the pull-up resistor. When a switch is closed, the corresponding peripheral data line falls to logic 0. In this application, both sides of the PIA act as input ports. Since they are automatically set up as inputs during reset, there is little to be done during initialization. Of course, bits 2 of the control registers must be set to 1 so that the input data from the keyboard can be read from addresses  $4008_{16}$  and  $400A_{16}$ .

The problems associated with decoding the keyboard are the same as those discussed earlier. Because this keyboard does not use interrupts, the MPU must scan the keyboard at regular intervals. Typically, it would read from addresses  $4008_{16}$  and  $400A_{16}$  several times each second.

The MPU detects a switch closure by comparing the input data with  $FF_{16}$ . If the input data is anything other than  $FF_{16}$ , one (or more) of the switches is closed.

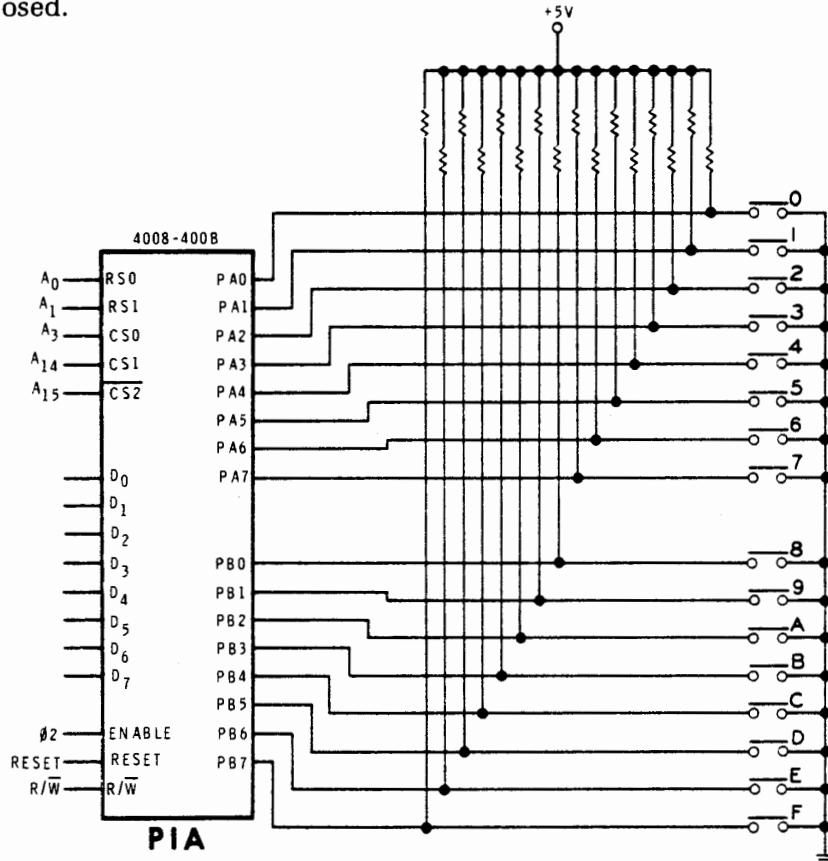


Figure 8-21  
Using the PIA to monitor a keyboard.

The MPU overcomes the switch bounce problem in the same way discussed earlier. Also, the problems of rejecting multiple switch closures and producing an equivalent binary code can be accomplished using the same techniques discussed previously in this unit.

## Decoding a Switch Matrix

The method shown in Figure 8-21 is a very straightforward technique of decoding switches. Using one PIA, this technique can handle up to 16 switches. There are other techniques that use the PIA to greater advantage. An example is shown in Figure 8-22. Here again, the PIA is handling 16 switches. However, this time only one side of the PIA is used.

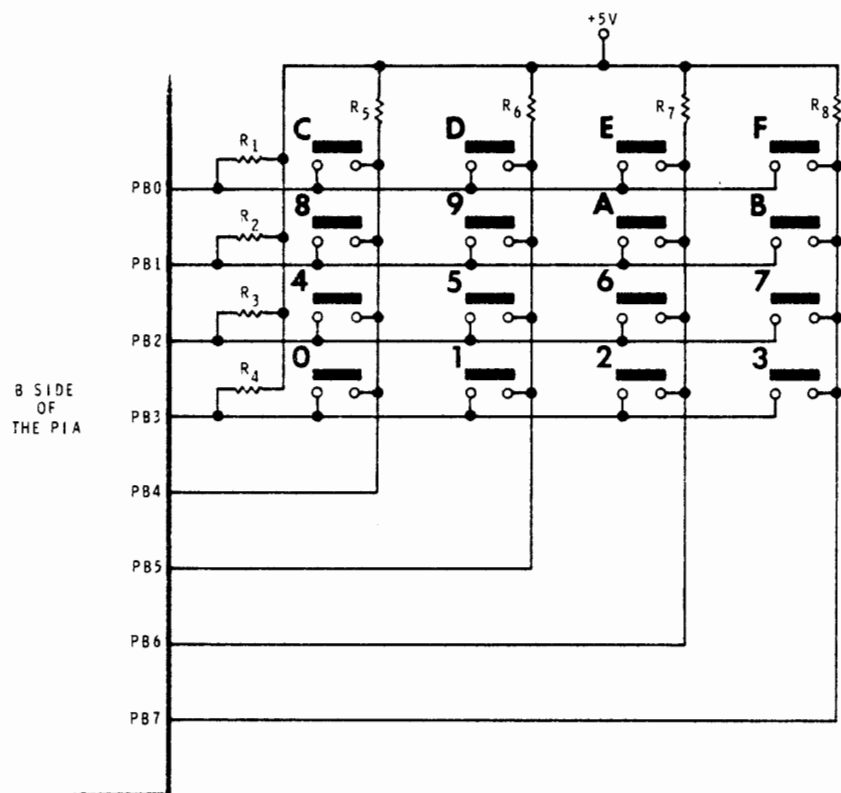


Figure 8-22  
One side of the PIA can handle up to 16 switches.

The 16 switches are arranged in a 4 by 4 matrix. The B side of the PIA is used to interface with the switches. Lines PB0 through PB3 are configured as input lines while PB4 through PB7 are configured as output lines.

With no switches closed, PB0 through PB3 are pulled up to logic 1 by resistors  $R_1$  through  $R_4$ . PB0 monitors switches C, D, E, and F. When all four switches are open, PB0 is at logic 1. Even with one of the switches closed, PB0 will still be at logic 1 if output lines PB4 through PB7 are at logic 1. And, these output lines (PB4 through PB7) are normally held at logic 1.

Periodically, the MPU scans the keyboard to see if any switch has been closed. It does this by applying logic 0 to one of the output lines and then checking for a logic 0 at one of the input lines.

A typical procedure might look like this. When the PIA is initialized, PB0 through PB3 are set up as inputs while PB4 through PB7 are set up as outputs. The MPU scans the keyboard in this manner. PB4 is reset to 0 by storing  $EF_{16}$  in output register B. Next, the B side is read out. If switch 0, 4, 8, or C is closed, its corresponding PIA input line will be low. For example, if switch 8 is closed, the 0 at PB4 will pull PB1 low. By examining lines PB0 through PB3, the MPU can tell which switch (if any) is closed.

If no switch is closed in the first column, PB5 is reset to 0 by storing  $DF_{16}$  in output register B. The MPU can now check to see if switch 1, 5, 9, or D is closed.

The technique just described allows the MPU to handle a large number of switches with a single PIA. Using both sides of the PIA, the MPU could handle an 8 by 8 matrix of  $64_{10}$  switches. This technique will be explored in more detail in an interfacing experiment.

## Self-Test Review

21. Refer to Figure 8-19. Write a short program that will initialize the PIA in the proper configuration.
22. Refer to Figure 8-19. Which side of the PIA determines which display is selected?
23. Refer to Figure 8-19. Assume that the PIA has been properly initialized. Write a simple program segment that will display the numeral 4 on display number 7.
24. Refer to Figure 8-21. In order for this scheme to work, both sides of the PIA must be configured as \_\_\_\_\_.
25. Refer to Figure 8-21. How does the MPU tell that switch 9 is closed?
26. How can the B side of the PIA be set up as shown in Figure 8-22?
27. Refer to Figure 8-22. How does the MPU tell that switch 7 is closed?
28. If both sides of the PIA are used, how many switches can be handled using the matrix technique?

## ANSWERS

21. Assuming that the PIA was initially reset, a typical initialization routine would be:

```
LDAA    #FF
STAA    4004
STAA    4006
LDAA    #04
STAA    4005
STAA    4007
```

22. The B side.

23. Keep in mind that  $Q_7$  must conduct and that pins b, c, g, and f must be high. Thus, the following instructions could be used:

```
LDAA    #80
STAA    4006
LDAA    #99
STAA    4004
```

24. inputs.

25. The MPU reads in data from both sides of the PIA and compares this data with several different bit patterns. If switch 9 is closed, the bit pattern from the B side will be  $FD_{16}$ .

26. The output register of the B side is configured in this manner during the initialization process by storing  $OF_{16}$  in the B side data direction register.

27. The MPU periodically resets line PB7 to 0. If PB2 is subsequently found to be 0, the MPU knows that switch 7 is closed.

28. Up to  $64_{10}$ .

## INTERFACING EXPERIMENTS

The interfacing experiments are included in Unit 10 of this course. Go to Unit 10 and perform experiments 5 through 9. Some of these experiments are quite involved and you should not attempt more than one experiment per sitting.



## UNIT EXAMINATION

1. In some microcomputer systems, input data from a keyboard is read repeatedly before it is accepted.
  - A. This speeds up the MPU.
  - B. This prevents two keys from being read at once.
  - C. This overcomes contact bounce.
  - D. All the above.
  
2. Refer to the program shown in Figure 8-5. Assume that key 6 is depressed. What hex number will be in accumulators A and B after the COMB instruction at line 18 is executed?
  - A. ACCB =  $40_{16}$ , ACCA =  $00_{16}$ .
  - B. ACCB =  $60_{16}$ , ACCA =  $00_{16}$ .
  - C. ACCB =  $06_{16}$ , ACCA =  $00_{16}$ .
  - D. ACCB =  $00_{16}$ , ACCA =  $06_{16}$ .
  
3. The purpose of the PIA is to:
  - A. Simplify the problem of interfacing the MPU to the ROM.
  - B. Simplify the problem of interfacing the MPU to the RAM.
  - C. Simplify the problem of interfacing the MPU to input/output devices.
  - D. Act as a universal input/output device.
  
4. The advantage of the PIA over conventional combinational logic circuits is:
  - A. Generally, fewer IC's are required.
  - B. The PIA is more flexible since its characteristics can be changed by the program.
  - C. In many cases, the PIA requires no separate address decoder.
  - D. All the above.
  
5. The A side of the PIA is selected when:
  - A. RS0 = 0.
  - B. RS0 = 1.
  - C. RS1 = 0.
  - D. RS1 = 1.
  
6. To select output register A of the PIA:
  - A. RS0 must be 0 and bit 2 of control register A must be 0.
  - B. RS0 must be 1 and bit 2 of control register A must be 0.
  - C. RS0 must be 0 and bit 2 of control register A must be 1.
  - D. RS0 must be 1 and bit 2 of control register A must be 1.

7. When the PIA is reset:
- A. Both sides are configured as outputs.
  - B. Both sides are configured as inputs.
  - C. The A side is configured as outputs while the B side is configured as inputs.
  - D. The B side is configured as outputs while the A side is configured as inputs.
8. Refer to Figure 8-19. Which of the following sequences will cause display 4 to display the number 1.
- |    |           |    |           |    |           |    |           |
|----|-----------|----|-----------|----|-----------|----|-----------|
| A. | LDAA #01  | B. | LDAA #9F  | C. | LDAA #60  | D. | LDAA #10  |
|    | STAA 4004 |    | STAA 4004 |    | STAA 4004 |    | STAA 4004 |
|    | LDAA #04  |    | LDAA #10  |    | LDAA #10  |    | LDAA #04  |
|    | STAA 4006 |    | STAA 4006 |    | STAA 4006 |    | STAA 4006 |
9. Refer to Figure 8-19. All displays can be blanked by storing:
- A. FF at address 4004.
  - B. 00 at address 4006.
  - C. FF at address 4004 and 00 at address 4006.
  - D. Any of the above.
10. Refer to Figure 8-22. An indication that switch A is closed is:
- A. PB1 goes low when PB6 goes low.
  - B. PB0 through PB4 are all high.
  - C. PB1 goes low when PB6 goes high.
  - D. PB0 goes low when PB6 goes low.



# Individual Learning Program

## MICROPROCESSORS

*Unit 9*

### PROGRAMMING EXPERIMENTS

EE-3401

HEATH COMPANY  
BENTON HARBOR, MICHIGAN 49022

Copyright © 1977  
Heath Company  
All Rights Reserved  
Printed in the United States of America

## CONTENTS

Introduction .....	Page 9-3
Experiment 1. Binary/Decimal Training Program .....	Page 9-4
Experiment 2. Hexadecimal/Decimal Training Program ....	Page 9-11
Experiment 3. Straight Line Programs .....	Page 9-19
Experiment 4. Arithmetic and Logic Instructions .....	Page 9-35
Experiment 5. Program Branches .....	Page 9-45
Experiment 6. Additional Instructions .....	Page 9-80
Experiment 7. New Addressing Modes .....	Page 9-100
Experiment 8. Arithmetic Operations .....	Page 9-109
Experiment 9. Stack Operations .....	Page 9-119
Experiment 10. Subroutines .....	Page 9-127

## UNIT 9

# PROGRAMMING EXPERIMENTS

### INTRODUCTION

This Unit contains ten programming experiments that are to be run on the Microprocessor Trainer. At the end of Units 1 through 6, you will be instructed to perform one or more of these experiments. Do not confuse these with the Interfacing Experiments which are in Unit 10.

The early programs given in this Manual are extremely simple. The later programs are more complex, but you will be able to accomplish them as you become familiar with the instruction set and programming techniques. Before you finish this course, you will be writing programs that will turn the trainer into a clock, a musical instrument, a digital voltmeter, etc.

When you complete an experiment, return to the activity guide of the unit that directed you to the experiment. This is important because you will be jumping from one point to another quite frequently.

# Experiment 1

## **BINARY/DECIMAL TRAINING PROGRAM**

### **OBJECTIVES:**

- To improve your ability to convert binary numbers to their decimal equivalent.*
- To improve your ability to convert decimal numbers to their binary equivalent.*
- To present the proper procedure for entering a program into the ET-3400 Microprocessor Trainer.*
- To demonstrate the versatility of the ET-3400 Microprocessor Trainer and microprocessors in general.*

## **Introduction**

In Unit 1, you were introduced to the binary number system. As you proceed through this course, you will find the need to convert binary numbers to decimal, and decimal numbers to binary. To improve your ability to make these conversions, you will enter a program into the Microprocessor Trainer to allow it to act as your instructor. In the first half of this experiment, you will use the Trainer to practice binary-to-decimal conversion.

When you use the Trainer, carefully follow all of the operating instructions. A microprocessor can only perform properly if it is programmed properly. However, you do not need programming experience at this time; just follow the instructions provided in this experiment. Do not worry about what you are entering.

The Trainer Manual contains a great amount of useful information in the Operation Section. You should review that section before you proceed with this experiment.

If your Trainer has been modified for use with the Heathkit Memory I/O Accessory, unplug the Trainer from the AC wall receptacle. Disconnect the 40-pin plug that connects the Trainer to the Memory I/O Accessory.

If your Trainer is Model number ET-3400, reinstall the 2112 RAM IC's at IC14 through IC17 before starting the experiments in this unit.

If your Trainer is model number ET-3400A, reinstall the 2114 RAM IC's at IC14 and IC15 before starting the experiments in this unit.





## Procedure

1. Plug in the Trainer and push the POWER switch on. Then momentarily press the RESET key. The display should show CPU UP.
2. Push the AUTO (automatic) key. Displays H, I, N, and Z will show "prompt" characters (bottom segment of each digit illuminated), and displays V and C will show Ad. NOTE: The letters identifying each display are located near their bottom right corners.
3. Push the 0 key three times. 0's will appear in displays H, I, and N.
4. Push, but do not release the 0 key. A 0 will appear in display Z. Now release the key. The 0 will not change, but displays V and C will now show prompt characters.

NOTE: The Trainer is now ready to receive program data. If you make a data error while entering the program, do not attempt to correct the error; continue programming. Any errors will be located and corrected when you examine your program.

5. Using the Trainer keys, enter the Binary-to-Decimal training program shown in Figure 9-1. At each address specified, press the appropriate inst/data (program instruction or data) number keys (most significant number first). Displays V and C will show the inst/data word you have entered. Note that as you release the second data key, address displays H, I, N, and Z will increment (count up one), and displays V and C will again show prompt characters. When you get to the end of the program, press the RESET key as indicated.

ADDRESS	INST DATA	ADDRESS	INST DATA	ADDRESS	INST DATA
0000	00*	0029	02	0052	00
0001	00*	002A	08	0053	3B
0002	BD	002B	00	0054	4F
0003	FC	002C	00	0055	DB
0004	BC	002D	00	0056	BD
0005	BD	002E	80	0057	00
0006	FE	002F	BD	0058	69
0007	52	0030	FC	0059	7E
0008	5E	0031	BC	005A	00
0009	FE	0032	BD	005B	02
000A	7C	0033	FE	005C	BD
000B	00	0034	09	005D	FE
000C	01	0035	97	005E	52
000D	B6	0036	00	005F	00
000E	C0	0037	BD	0060	00
000F	03	0038	00	0061	15
0010	01	0039	69	0062	9D
0011	46	003A	5F	0063	BD
0012	25	003B	84	0064	00
0013	F6	003C	F0	0065	69
0014	CE	003D	27	0066	7E
0015	00	003E	07	0067	00
0016	00	003F	80	0068	14
0017	DF	0040	10	0069	86
0018	F2	0041	CB	006A	02
0019	BD	0042	9A	006B	CE
001A	FD	0043	4D	006C	00
001B	93	0044	26	006D	00
001C	B6	0045	F9	006E	09
001D	C0	0046	96	006F	26
001E	06	0047	00	0070	FD
001F	01	0048	84	0071	4A
0020	46	0049	0F	0072	26
0021	25	004A	15	0073	F7
0022	F9	004B	90	0074	96
0023	BD	004C	01	0075	01
0024	FC	004D	26	0076	84
0025	BC	004E	0D	0077	3F
0026	BD	004F	BD	0078	97
0027	FE	0050	FE	0079	01
0028	52	0051	52	007A	96
				007B	00
				007C	39
					RESET

\*This data may change randomly.

Figure 9-1  
Binary-to-decimal training program.

6. Now that you have entered the Binary-to-Decimal training program, you must examine the data for errors. Use the following sequence to examine the data and correct any errors.
  - A. Press the EXAM (examine) key. Note that the display is now asking for a 4-digit address ( \_ \_ \_ \_ Ad.)
  - B. Enter the beginning address of the program (0000). As soon as the last address digit is entered, displays V and C show the contents of that memory location. NOTE: The address is a memory location in the Trainer.
  - C. Now compare the displayed address and data with the address and inst/data columns in the program.
  - D. If the displayed data is incorrect, press the CHAN (change) key. The data displays will now show prompt characters. Enter the correct data.
  - E. Press the FWD (forward) key. The address will increment and the data for that memory location will be shown. Correct the data if necessary.
  - F. Continue to step through the program with the FWD key, and correct data as necessary, until you reach the end of the program. It is not necessary to examine or modify the memory beyond address 007C since it will have no effect on the program.
7. Press the RESET key.
8. Press the DO key, then enter address 0002. The display should show GO. If the display shows a different number or word, or goes blank, your program contains an error. Repeat steps 6 through 8.
9. Press the F key. A 6-bit binary number should appear in the display. This is a random number and should change in value when you are told to "GO" next time.
10. Examine the binary number and determine its decimal value. Then press the D key. Two prompt characters should appear in the display.

11. Enter the decimal value of the binary number previously displayed (most significant digit first.) For values less than 10, enter a 0 before you enter the value. After a short period of time, the Trainer will indicate whether or not your answer is correct.
12. If your answer was correct, the Trainer will display YES. A moment later, the word GO will replace the decimal number.

If your answer was incorrect, the Trainer will display NO. The same binary number will again be displayed. Determine and enter the decimal value as described in steps 10 and 11.

13. Refer again to steps 9 through 12 and practice converting binary numbers to their decimal equivalent. You should obtain 10 correct answers in succession before you continue with this experiment.

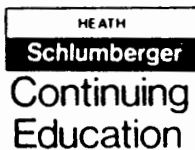
## Discussion

Now that you have used the Trainer and its microprocessor, you have accomplished three objectives. First, you are becoming proficient in binary-to-decimal conversion. Second, you have been introduced to the correct method for entering, examining, and modifying a program. Third, you have been shown how a simple set of instructions can produce a powerful training aid. However, you should remember, a microprocessor can only perform what you tell it. One incorrect instruction can produce totally unexpected results.

Now, reprogram the Trainer for decimal-to-binary instruction. Since you will be using the same memory locations used in the first half of this experiment, the Binary-To-Decimal program will disappear.

## Procedure (Continued)

14. Press the RESET key.
15. Press the AUTO key, and enter address 0000.
16. Using the Trainer keys, enter the Decimal-to-Binary training program shown in Figure 9-2.
17. Now that you have entered the Decimal-to-Binary program, press the EXAM key and enter address 0000.



HEATH COMPANY  
BENTON HARBOR, MICHIGAN 48022

## **IMPORTANT NOTICE**

Dear Customer:

Please replace Page 9-9 of your Heath Continuing Education program — EE-3401 Micro-processors — with the enclosed page.

We are sorry for any inconvenience this may have caused you.

Thank you,

HEATH COMPANY



ADDRESS	INST/DATA	ADDRESS	INST/DATA	ADDRESS	INST/DATA
0000	00*	0033	4F	0066	00
0001	CE	0034	E6	0067	00
0002	C1	0035	00	0068	80
0003	6F	0036	C5	0069	7E
0004	BD	0037	10	006A	00
0005	FE	0038	27	006B	01
0006	50	0039	03	006C	BD
0007	5E	003A	AB	006D	FE
0008	FE	003B	03	006E	52
0009	96	003C	19	006F	15
000A	00	003D	56	0070	1D
000B	8B	003E	24	0071	00
000C	01	003F	03	0072	00
000D	19	0040	AB	0073	00
000E	81	0041	06	0074	80
000F	63	0042	19	0075	BD
0010	23	0043	08	0076	00
0011	01	0044	8C	0077	7E
0012	4F	0045	00	0078	BD
0013	97	0046	88	0079	FC
0014	00	0047	26	007A	BC
0015	B6	0048	EB	007B	7E
0016	CO	0049	BD	007C	00
0017	03	004A	00	007D	1C
0018	01	004B	7E	007E	36
0019	46	004C	BD	007F	BD
001A	25	004D	FC	0080	00
001B	ED	004E	BC	0081	8E
001C	96	004F	D6	0082	32
001D	00	0050	00	0083	01
001E	BD	0051	11	0084	39
001F	FE	0052	26	0085	00
0020	20	0053	18	0086	00
0021	B6	0054	BD	0087	00
0022	CO	0055	FE	0088	32
0023	06	0056	52	0089	08
0024	01	0057	00	008A	02
0025	46	0058	00	008B	16
0026	25	0059	00	008C	04
0027	F9	005A	3B	008D	01
0028	BD	005B	4F	008E	86
0029	FC	005C	DB	008F	02
002A	BC	005D	BD	0090	CE
002B	C6	005E	00	0091	00
002C	03	005F	7E	0092	00
002D	CE	0060	CE	0093	09
002E	00	0061	C1	0094	26
002F	85	0062	3F	0095	FD
0030	BD	0063	BD	0096	49
0031	FD	0064	FE	0097	26
0032	25	0065	50	0098	F7
				0099	39
					RESET

\*This data may change randomly

Figure 9-2  
 Decimal-to-binary training program.

18. Using the FWD key, compare the Trainer memory contents with the program address and inst/data listing. If you must correct any data, press the CHAN key and enter the proper data.
19. After you have checked the program, press the RESET key.
20. Press the DO key, then enter address 0001. The display should show GO. If the display shows a different number or word, or goes blank, your program contains an error. Repeat steps 17 through 20.
21. Press the F key. A 2-digit decimal number should appear in the display, next to the word GO. This is a random number and should change in value when you are told to "GO" next time.
22. Examine the decimal number and determine its binary value. Then press the D key. Six prompt characters should appear in the display.
23. Enter the binary value of the decimal number previously displayed, beginning with the most significant bit (MSB). If the decimal value is less than 32, be sure to enter any leading zeros. NOTE: Although the program will accept any number combination, you should use only 1's and 0's.
24. If your answer was correct, the Trainer will display YES a short time after you enter the last binary bit. A moment later, the Trainer will display GO.  
  
If your answer was incorrect, the Trainer will display NO a short time after you enter the last binary bit. A moment later, the same decimal number will be displayed again. Determine and enter the binary value as described in steps 22 and 23.
25. Refer again to steps 21 through 24 and practice converting decimal numbers to their binary equivalent. You should obtain 10 correct answers in succession before you continue with this experiment.

## Discussion

In this half of the experiment, you were given further experience in programming with the ET-3400 Microprocessor Trainer. You also improved your ability to readily translate decimal numbers into binary. This ability will become very useful as you progress through the Microprocessor Course.



ADDRESS	INST/DATA	ADDRESS	INST/DATA	ADDRESS	INST/DATA
0000	00*	0033	4F	0062	3F
0001	CE	0034	E6	0063	BD
0002	C1	0035	00	0064	FE
0003	6F	0036	C5	0065	50
0004	BD	0037	10	006A	00
0005	FE	0038	27	006B	01
0006	50	0039	03	006C	BD
0007	5E	003A	AB	006D	FE
0008	FE	003B	03	006E	52
0009	96	003C	19	006F	15
000A	00	003D	56	0070	1D
000B	8B	003E	24	0071	00
000C	01	003F	03	0072	00
000D	19	0040	AB	0073	00
000E	81	0041	06	0074	80
000F	63	0042	19	0075	BD
0010	23	0043	08	0076	00
0011	01	0044	8C	0077	7E
0012	4F	0045	00	0078	BD
0013	97	0046	88	0079	FC
0014	00	0047	26	007A	BC
0015	B6	0048	EB	007B	7E
0016	CO	0049	BD	007C	00
0017	03	004A	00	007D	1C
0018	01	004B	7E	007E	36
0019	46	004C	BD	007F	BD
001A	25	004D	FC	0080	00
001B	ED	004E	BC	0081	8E
001C	96	004F	D6	0082	32
001D	00	0050	00	0083	01
001E	BD	0051	11	0084	39
001F	FE	0052	26	0085	00
0020	20	0053	18	0086	00
0021	B6	0054	BD	0087	00
0022	CO	0055	FE	0088	32
0023	06	0056	52	0089	08
0024	01	0057	00	008A	02
0025	46	0058	00	008B	16
0026	25	0059	00	008C	04
0027	F9	005A	3B	008D	01
0028	BD	005B	4F	008E	86
0029	FC	005C	DB	008F	02
002A	BC	005D	BD	0090	CE
002B	C6	0066	00	0091	00
002C	03	0067	00	0092	00
002D	CE	0068	80	0093	09
002E	00	0069	7E	0094	26
002F	85	005E	00	0095	FD
0030	BD	005F	7E	0096	49
0031	FD	0060	CE	0097	26
0032	25	0061	C1	0098	F7
				0099	39
					RESET

\*This data may change randomly

Figure 9-2  
 Decimal-to-binary training program.

18. Using the FWD key, compare the Trainer memory contents with program address and inst/data listing. If you must correct any data, press the CHAN key and enter the proper data.
19. After you have checked the program, press the RESET key.
20. Press the DO key, then enter address 0001. The display should show GO. If the display shows a different number or word, or goes blank, your program contains an error. Repeat steps 17 through 20.
21. Press the F key. A 2-digit decimal number should appear in the display, next to the word GO. This is a random number and should change in value when you are told to "GO" next time.
22. Examine the decimal number and determine its binary value. Then press the D key. Six prompt characters should appear in the display.
23. Enter the binary value of the decimal number previously displayed, beginning with the most significant bit (MSB). If the decimal value is less than 32, be sure to enter any leading zeros. NOTE: Although the program will accept any number combination, you should use only 1's and 0's.
24. If your answer was correct, the Trainer will display YES a short time after you enter the last binary bit. A moment later, the Trainer will display GO.  
  
If your answer was incorrect, the Trainer will display NO a short time after you enter the last binary bit. A moment later, the same decimal number will be displayed again. Determine and enter the binary value as described in steps 22 and 23.
25. Refer again to steps 21 through 24 and practice converting decimal numbers to their binary equivalent. You should obtain 10 correct answers in succession before you continue with this experiment.

## Discussion

In this half of the experiment, you were given further experience in programming with the ET-3400 Microprocessor Trainer. You also improved your ability to readily translate decimal numbers into binary. This ability will become very useful as you progress through the Microprocessor Course.

## Experiment 2

### HEXADECIMAL/DECIMAL TRAINING PROGRAM

#### OBJECTIVES:

*To practice the conversion of decimal numbers to their hexadecimal equivalent.*

*To practice the conversion of hexadecimal numbers to their decimal equivalent.*

#### Introduction

Binary numbers are used in all microprocessors to represent data and instructions. But binary numbers are difficult to work with . . . especially when the number contains  $8_{10}$  bits or more. To simplify programming, microprocessor designers usually use other number systems, like octal or hexadecimal, to represent binary data. Both octal and hexadecimal are just shorthand notations of binary numbers. Although the numbers are entered in hexadecimal or octal, the microprocessor "sees" them as binary. This simplifies programming.

For example, the binary number  $10011111_2$  requires eight key closures for entry. Fortunately, this same number can be represented in hexadecimal as  $9F_{16}$  and requires only two key closures for entry. Fewer key closures means less programming errors and more efficient programming.

Your Microprocessor Trainer is based on the hexadecimal number system. You probably noticed this when you loaded the programs in the previous experiment; all instructions were coded in hexadecimal. The Microprocessor Trainer normally displays data in hexadecimal form. Of course, special programs allow the Trainer to accept binary or decimal numbers, as you saw in the first experiment. However, these special programs waste a portion of the microprocessors potential power and aren't necessary because you can make the conversion from decimal to hexadecimal with a little practice. That's the purpose of this experiment . . . to sharpen your conversion skills.

Again, you will use the Microprocessor Trainer for this purpose. First, you'll enter a program that allows you to practice conversion from decimal to hexadecimal. Then you'll load the second program that reverses the process. You'll find that it's not as difficult as it might appear.

Now briefly review decimal-to-hexadecimal conversion. Initially, it's helpful to make up a chart of decimal numbers and their hexadecimal equivalents, as shown here.

DECIMAL	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
HEXADECIMAL	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

Recall that hexadecimal is a base  $16_{10}$  number system. Both systems use identical numbers from 0 through 9. However, at decimal number 10, the hexadecimal system shifts to characters of the alphabet, as shown by the letters A through F. Conversion of a decimal number to its hexadecimal equivalent is a simple process where the decimal number is repeatedly divided by  $16_{10}$ , with the remainder producing the equivalent hexadecimal number. This example will use only 2-digit numbers, since that's what you'll be converting in this experiment.

Suppose you want to convert  $92_{10}$  to hexadecimal. The first step is to divide  $92_{10}$  by  $16_{10}$  as shown below.

$$\begin{array}{r}
 5 \\
 16 \overline{) 92} \\
 \underline{- 80} \\
 \text{Remainder} \quad 12_{10} = C_{16} \leftarrow \text{LSD}
 \end{array}$$

The quotient is 5, but remember, we aren't concerned with this at the moment. We're interested in the remainder, in this case  $12_{10}$ , because it forms the LSD of the equivalent hexadecimal number. Now, refer to the chart and find that  $12_{10} = C_{16}$  and write this down as the LSD of the hex equivalent. The next step is to take the quotient of the previous division, in this case  $5_{10}$ , and divide it by  $16_{10}$ , as shown below.

$$\begin{array}{r}
 0 \\
 16 \overline{) 5} \\
 \underline{- 0} \\
 \text{Remainder} \rightarrow 5_{10} = 5_{16} \leftarrow \text{MSD}
 \end{array}$$

Of course, the quotient of this division is 0, signifying that the remainder,  $5_{10}$ , is the MSD of the hexadecimal number. Checking the chart, you find that  $5_{10} = 5_{16}$ . Combining the MSD ( $5_{16}$ ) and LSD ( $C_{16}$ ), you find that the hex equivalent of  $92_{10}$  is  $5C_{16}$ . You'll find that, after you've made the conversion a few times, you'll be able to do them in your head. You'll get that practice in this experiment.

## Procedure

1. Turn on the Trainer and press the RESET key.
2. Press AUTO and then enter address 0000.
3. Now enter the Decimal-to-Hexadecimal training program, shown in Figure 9-3, into the Trainer. When you've entered the last program instruction press the RESET key as shown at the end of the program.

ADDRESS	INST/DATA	ADDRESS	INST/DATA	ADDRESS	INST/DATA	ADDRESS	INST/DATA
0000	00*	0024	BD	0048	52	006C	00
0001	CE	0025	FE	0049	00	006D	00
0002	C1	0026	52	004A	3B	006E	00
0003	6F	0027	08	004B	4F	006F	00
0004	BD	0028	08	004C	DB	0070	00
0005	FE	0029	00	004D	BD	0071	80
0006	50	002A	00	004E	00	0072	39
0007	5E	002B	00	004F	63	0073	86
0008	FE	002C	80	0050	7E	0074	02
0009	96	002D	BD	0051	00	0075	CE
000A	00	002E	FC	0052	01	0076	00
000B	8B	002F	BC	0053	BD	0077	00
000C	01	0030	BD	0054	FE	0078	09
000D	19	0031	FE	0055	52	0079	26
000E	97	0032	09	0056	00	007A	FD
000F	00	0033	36	0057	00	007B	4A
0010	B6	0034	4F	0058	15	007C	26
0011	CO	0035	D6	0059	9D	007D	F7
0012	03	0036	00	005A	BD	007E	39
0013	46	0037	CO	005B	00		RESET
0014	25	0038	10	005C	63		
0015	F3	0039	25	005D	BD		
0016	96	003A	04	005E	FC		
0017	00	003B	8B	005F	BC		
0018	BD	003C	0A	0060	7E		
0019	FE	003D	20	0061	00		
001A	20	003E	F8	0062	16		
001B	B6	003F	CB	0063	36		
001C	CO	0040	10	0064	BD		
001D	06	0041	1B	0065	00		
001E	46	0042	33	0066	73		
001F	25	0043	11	0067	32		
0020	FA	0044	26	0068	01		
0021	BD	0045	OD	0069	BD		
0022	FC	0046	BD	006A	FD		
0023	BC	0047	FE	006B	8D		

\* This data may change randomly

Figure 9-3

Decimal-to-hexadecimal training program

4. Check the stored program by first pressing the EXAM key and then entering address 0000. Now use the FWD key to step through the program, comparing the contents of memory with the program in Figure 9-3. Remember, the four left-most digits of the display represent the memory address and the two digits at the right are the contents of memory that should correspond with the INST/DATA listing of the program. If you find a mistake, correct it by first pressing the CHAN key and then entering the proper data.
5. When you're satisfied that the program is correct, press the RESET key.
6. Now it's time to execute the program. Do this by pressing the DO key and then entering address 0001. The word "GO" should now appear in the two left-most digits. If the display is blank, or if other numbers or letters appear, there is an error in the program and steps 4 and 5 should be repeated.
7. Now press the F key. A 2-digit "decimal" number will appear on the display. The Trainer is asking you to convert this decimal number to its hexadecimal equivalent. Therefore, examine the decimal number and then convert it to hexadecimal.
8. Enter your answer by first pressing key D. Two prompt characters will appear in the left-most digits. Now enter your hexadecimal number.

If you respond correctly, the Trainer will display "YES" for a short period and then give you another "GO." Pressing the F key will cause another random number to be displayed.

An incorrect response will result in the word "NO" on the display. After a short delay, the original decimal number will reappear and you should try the conversion process again. This cycle continues until you arrive at the correct answer.

9. Repeat steps 7 and 8, practicing conversion until you're confident of your ability. A good guideline to follow is . . . . when you answer 10 consecutive queries correctly, you're probably proficient.

## Discussion

As you worked through the exercises in this experiment, you probably developed your own shorthand method of conversion. After a few queries, you probably found that you didn't need the decimal-to-hexadecimal conversion chart any longer . . . you had the chart committed to memory. Perhaps you noticed that when  $16_{10}$  is divided into the 2-digit decimal numbers used in this experiment, the resulting quotient always equals the MSD of the hexadecimal equivalent. Naturally, the remainder is the LSD. However, this only works for decimal numbers less than  $159_{10}$ . For larger numbers, the procedure studied earlier must be used.

Since the Microprocessor Trainer displays data in hexadecimal and we naturally think in decimal, the conversion process must be reversed to interpret output data from the Trainer. For example, if the Trainer is programmed to add the numbers  $1A_{16}$  and  $9B_{16}$ , the result  $B5_{16}$  will be displayed. This hexadecimal number means very little. To understand the result, you must convert the sum ( $B5_{16}$ ) to its decimal equivalent ( $181_{10}$ ). Now the answer is clear.

Several methods can be used to change hexadecimal numbers to decimal. One process uses double conversion; first, the hexadecimal number is reduced to its binary equivalent; next, the resulting binary number is transformed into the resulting decimal equivalent.

Another, more commonly used method, is to use positional notation, inherent in any number system, and multiply each digit by its weighted value and then add the products. For example, the decimal equivalent of the hex number  $11_{16}$  is derived as shown below:

	Assign Weights: $16^1$	$16^0$ Positional Weights
	1	1
Weight $\times$ Digit: $1 \times 16^1 = 16$	$\leftarrow$	$\rightarrow 1 \times 16^0 = 1$
Add Products:	$16 + 1 = 17$	
Final Result:	$11_{16} = 17_{10}$	

The first step is to assign positioned weights to each digit. Since the number is hexadecimal, each position represents a power of  $16_{10}$ . Next, multiply each digit by its positional weight. Finally, add the products. The resulting sum is the decimal equivalent. Therefore, as shown in the example,  $11_{16}$  is equal to  $17_{10}$ .

Now try a problem that's a bit more difficult . . . converting  $6B_{16}$  to decimal. To begin with, this expression hardly looks like a number. Instead, it's a combination of a number and a letter. However, the notation at the bottom of the expression denotes a base 16 number so we know it's hexadecimal. The translation process is almost identical to the previous example. The only difference being that the hexadecimal "letter" must be changed to decimal before it can be multiplied by the positional weight. The conversion process is shown below.

Assign Weights:	$16^1$	$16^0$
	6	B
Convert to Decimal:	$6_{16} = 6_{10}$	$B_{16} = 11_{10}$
Weight $\times$ Digit:	$6 \times 16^1 = 96$	$11 \times 16^0 = 11$
Add Products:	$96 + 11 = 107$	
Final Result:	$6B_{16} = 107_{10}$	

Again, we begin by assigning positional weights to each digit. However, now the second step is to convert the hexadecimal characters to decimal numbers. Recall that  $6_{16}$  is equal to  $6_{10}$  and that  $B_{16}$  equals  $11_{10}$ . Now multiply the weight by the decimal numbers, add the products and obtain the final result. As shown, the decimal equivalent of  $6B_{16}$  is  $107_{10}$ .

In the next section of this experiment, you will load a hexadecimal-to-decimal training program in the Trainer and then practice hexadecimal-to-decimal conversion.

### Procedure (Continued)

10. Prepare to enter the new program by pressing the RESET key. Next press the AUTO key and then enter address 0000.
11. Refer to Figure 9-4 and enter the Hexadecimal-to-Decimal training program listed there. When you've entered all of the instructions, press the RESET key as indicated at the end of the program.
12. Check the program that you've loaded by pressing the EXAM key and then entering address 0000. Use the FWD key to step through the program, comparing the stored program with the program listing in Figure 9-4. Use the CHAN key to correct any errors that you find.

When you are satisfied that the program is correct, press the RESET key.



ADDRESS	INST/DATA	ADDRESS	INST/DATA	ADDRESS	INST/DATA	ADDRESS	INST/DATA
0000	00	0024	BD	0048	FE	006C	8D
0001	CE	0025	FC	0049	52	006D	00
0002	C1	0026	BC	004A	00	006E	00
0003	6F	0027	BD	004B	3B	006F	00
0004	BD	0028	FE	004C	4F	0070	00
0005	FE	0029	52	004D	DB	0071	00
0006	50	002A	08	004E	BD	0072	80
0007	5E	002B	08	004F	00	0073	39
0008	FE	002C	00	0050	64	0074	86
0009	96	002D	00	0051	7E	0075	02
000A	00	002E	00	0052	00	0076	CE
000B	4C	002F	80	0053	01	0077	00
000C	81	0030	BD	0054	BD	0078	00
000D	63	0031	FC	0055	FE	0079	09
000E	23	0032	BC	0056	52	007A	26
000F	01	0033	BD	0057	00	007B	FD
0010	4F	0034	FE	0058	00	007C	4A
0011	97	0035	09	0059	15	007D	26
0012	00	0036	5F	005A	9D	007E	F7
0013	B6	0037	80	005B	BD	007F	39
0014	CO	0038	10	005C	00		RESET
0015	03	0039	25	005D	64		
0016	46	003A	04	005E	BD		
0017	25	003B	CB	005F	FC		
0018	FO	003C	OA	0060	BC		
0019	96	003D	20	0061	7E		
001A	00	003E	F8	0062	00		
001B	BD	003F	8B	0063	19		
001C	FE	0040	10	0064	36		
001D	20	0041	1B	0065	BD		
001E	B6	0042	D6	0066	00		
001F	CO	0043	00	0067	74		
0020	06	0044	11	0068	32		
0021	46	0045	26	0069	01		
0022	25	0046	OD	006A	BD		
0023	FA	0047	BD	006B	FD		

Figure 9-4  
Hexadecimal-to-decimal training program

13. Now execute the program by first pressing the DO key and then entering address 0001. The word "GO" should appear on the display. The absence of this word indicates a programming error and you should go back and recheck the program as outlined in step 12.
14. Now press the F key. A 2-digit "hexadecimal" number will appear. The Trainer is asking for the decimal equivalent of this number. Convert the hexadecimal number into its decimal equivalent. Then enter your answer by pressing the D key. Two prompt characters will appear. Now enter your answer.

If your response is correct, the Trainer will display "YES." You can then continue these conversion exercises by again pressing the F key.

However, if your answer is incorrect, the Trainer will display "NO." After a short delay, the original hexadecimal number will reappear, and you can try again.

15. Continue the conversion training program until you are confident of your ability to change hexadecimal numbers to decimal numbers. The standard of ten correct conversions in a row is a good guideline.

## Discussion

The translation of hexadecimal numbers into decimal equivalent numbers is an important part of your training.

You will find this skill is extremely handy when you begin to write programs later in this course. Now you should be able to convert between hexadecimal and decimal numbers with ease. Perhaps you even developed your own shorthand methods for these translations. If so, use them. However, a word of caution . . . be sure they work for all numbers. As mentioned previously, some techniques work with small numbers, but not with large numbers.

## Experiment 3

### STRAIGHT LINE PROGRAMS

#### OBJECTIVES:

*To demonstrate the instructions presented in Unit 2 with simple programs.*

*To present three new instructions and use them in simple programs.*

*To demonstrate some programming pitfalls.*

*To demonstrate the difference between RAM and ROM.*

#### Introduction

Unit 2 introduced you to the basic microprocessor and its internal structure. You also learned six basic microprocessor instructions that are represented by 8-bit binary numbers called "op codes." Op codes allow you to use the microprocessor for data manipulation. Figure 9-5 lists the six instructions and their op codes. It also lists three new instructions that you will use in this experiment. These new instructions use the inherent addressing mode described in Unit 2.

This is the first experiment to introduce microprocessor instructions that you can identify. There are a number of Trainer keyboard commands that you must learn in order to examine and use the microprocessor instructions. The Trainer commands that you should know for this experiment are:

**DO** — Execute the program, beginning at the address specified after this key is pressed.

**EXAM** (examine) — Display the address and memory contents at the address specified after this key is pressed. Memory contents can be changed by pressing the **CHAN** key and entering new data.

**FWD** (forward) — Advance to the next memory location and display the contents.

**CHAN** (change) — Open the memory location being examined so that new data can be entered.

NAME	MNEMONIC	OPCODE	DESCRIPTION
Load Accumulator (Immediate)	LDA	1000 0110 <sub>2</sub> or 86 <sub>16</sub>	Load the contents of the next memory location into the accumulator.
Add (Immediate)	ADD	1000 1011 <sub>2</sub> or 8B <sub>16</sub>	Add the contents of the next memory location to the present contents of the accumulator. Place the sum in the accumulator.
Load Accumulator (Direct)	LDA	1001 0110 <sub>2</sub> or 96 <sub>16</sub>	Load the contents of the memory location whose address is given by the next byte into the accumulator.
Add (Direct)	ADD	1001 1011 <sub>2</sub> or 9B <sub>16</sub>	Add the contents of the memory location whose address is given by the next byte to the present contents of the accumulator. Place the sum in the accumulator.
Store Accumulator (Direct)	STA	1001 0111 <sub>2</sub> or 97 <sub>16</sub>	Store the contents of the accumulator in the memory location whose address is given by the next byte.
Halt (Inherent)	HLT	0011 1110 <sub>2</sub> or 3E <sub>16</sub>	Stop all operations.
Clear Accumulator (Inherent)	CLRA	0100 1111 <sub>2</sub> or 4F <sub>16</sub>	Reset all bits in the accumulator to 0.
Increment Accumulator (Inherent)	INCA	0100 1100 <sub>2</sub> or 4C <sub>16</sub>	Add 1 to the contents of the accumulator.
Decrement Accumulator (Inherent)	DECA	0100 1010 <sub>2</sub> or 4A <sub>16</sub>	Subtract 1 from the contents of the accumulator.

Figure 9-5  
Instructions used in Experiment 3.

**BACK** — Go back to the previous memory location and display the contents.

**AUTO** (automatic) — Open the memory location specified, after this key is pressed, so that data can be entered. After data has been entered, automatically advance to the next memory location and wait for data.

**SS** (single step) — Go to the address specified by the program counter and execute the instruction at that address. Wait at the next instruction.

**ACCA** (accumulator) — Display the contents of the accumulator when this key is pressed. Accumulator contents can be changed by pressing the **CHAN** key and entering new data.

**PC** (program counter) — Display the contents of the program counter. This points to the next location in memory that the microprocessor will “fetch” from. Program counter contents can be changed by pressing the **CHAN** key and entering the new address.

**RESET** — Clear any Trainer keyboard commands and display “CPU UP.” Memory contents and microprocessor contents are not disturbed.

You have access to all of these keyboard commands after the **RESET** key is pressed.

In this experiment, you will load some simple straight-line programs into the Trainer and examine how the microprocessor executes them. In its normal mode of operation, the microprocessor executes programs much too fast for a person to follow. It can execute hundreds of thousands of instructions each second. To allow us to witness the operation of the MPU, this high speed operation must be slowed down. The Microprocessor Trainer has a mode of operation that allows us to control the execution of single instructions. In this single-step mode, we can look at the contents of the accumulator, the program counter, and various memory locations, after each instruction is executed. In this way, we can follow exactly how the computer performs each step of the program. For this reason, you will use the single-step mode for most of the programs in this experiment.

## Procedure

1. Switch your Trainer on, and press the RESET key.
2. Your first program will use the immediate addressing mode to add two numbers. Press AUTO and enter starting address 0000. Then load the hex contents of the program listed in Figure 9-6.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	86	LDA	Load accumulator immediately with
0001	21	33 <sub>16</sub>	Operand 1.
0002	8B	ADD	Add to accumulator immediately with
0003	17	23 <sub>16</sub>	Operand 2.
0004	3E	HLT	Stop.

Figure 9-6

Addition of two numbers through the immediate addressing mode.

3. Press the RESET key, then examine your program to make sure it was properly entered. **Always** examine your program after it is entered.
4. Press the ACCA key and record the value \_\_\_\_\_. This is a random number since no data has been loaded.
5. Press the PC key, then change the contents of the program counter to 0000 (the starting address of your program).
6. Press the SS key. This lets the Trainer execute the first instruction. The display should show 00028b. 0002 represents the address of the next instruction; 8b is the next instruction.
7. Press the ACCA key and record the value \_\_\_\_\_. The first program instruction was LDA, and the next byte contained the data (operand) to be loaded, which is 21<sub>16</sub>. This should be the value you recorded in this step.
8. Press the PC key and record the value \_\_\_\_\_. This value points to the next memory location, which should be 0002.

You may have noted that the address 0002 and instruction 8b were displayed when you first pressed the SS key. This would seem to indicate that 8b was already fetched and the program counter should point to address 0003. However, the control program allows the Trainer to "look" at the next instruction.

9. Press the SS key and record the value \_ \_ \_ \_ \_ . The second instruction has been executed and the display should show the next instruction and its address.
10. Press the ACCA key and record the value \_ \_ . The second operand has been added to the first operand and the sum is stored in the accumulator.
11. Press the SS key. Note that the display does not change. This is because the next instruction was a halt instruction ( $3E_{16}$ ). The Trainer is preprogrammed to stop at a halt instruction. It also loses control of the single-step function when the halt instruction is implemented.
12. Enter the program (HEX contents) listed in Figure 9-7. Then examine the program to make sure it is properly entered.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	96	LDA	Load accumulator direct with operand 1 which is stored at this address.
0001	07	$07_{16}$	
0002	9B	ADD	Add to accumulator direct with operand 2 which is stored at this address.
0003	08	$08_{16}$	
0004	97	STA	Store the sum
0005	09	$09_{16}$	at this address.
0006	3E	HLT	Stop.
0007	20	$32_{10}$	Operand 1.
0008	17	$23_{10}$	Operand 2.
0009	00	00	Reserved for sum.

Figure 9-7

Addition of two numbers through the direct addressing mode.

13. Press the ACCA key and record the value \_ \_ . This is the value obtained in the previous program, a value you entered prior to this program, or a random value produced when you plugged in the Trainer.

14. Enter the program starting address into the program counter and single-step through the program. Record the specified information after each step.

Step 1 display \_ \_ \_ \_ \_.

ACCA \_ \_.

Step 2 display \_ \_ \_ \_ \_.

ACCA \_ \_.

Step 3 display \_ \_ \_ \_ \_.

ACCA \_ \_.

15. Examine address 0009. Its value is \_ \_ . This value should be identical to the value now stored in the ACCA.
16. Now compare your recorded data with the program in Figure 9-7. This will give you a general picture of how the microprocessor uses various instructions and data to perform a desired function.
17. Change the data in the ACCA and at address 0009 to FF, then execute the program with the DO key. This is done by depressing the DO key and then entering the address of the first instruction (0000). This allows the MPU to execute the program at its normal speed. After the program runs, you must press RESET to return control to the keyboard.
18. The data in the ACCA is \_ \_ and the data in address 0009 is \_ \_ . These should be the same and equal to the sum of the two operands.
19. The program counter contains the address \_ \_ \_ . This should be the address of the next memory location after the HLT instruction.



20. Now write a program of your own. Using the **direct** addressing mode, write a program that will multiply 4 times 4, by adding 4 to itself in three consecutive steps. The final answer should be held in the accumulator. After you write your program, enter it into the Trainer and execute it. Keep trying until it produces a final result of  $10_{16}$  (which is  $16_{10}$ ) in the accumulator.

One solution to the problem is shown in Figure 9-8. Yours should be similar, although not necessarily identical.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/DECIMAL CONTENTS	COMMENTS
0000	96	LDA	Load accumulator direct with
0001	09	$09_{16}$	operand 1 which is stored at this address.
0002	9B	ADD	Add to accumulator direct with
0003	09	$09_{16}$	operand 1 which is stored at this address.
0004	9B	ADD	Add to accumulator direct with
0005	09	$09_{16}$	operand 1 which is stored at this address.
0006	9B	ADD	Add to accumulator direct with
0007	09	$09_{16}$	operand 1 which is stored at this address.
0008	3E	HLT	Stop.
0009	04	$04_{10}$	Operand 1.

Figure 9-8

Multiplication of a number by another through multiple addition in the direct addressing mode.

21. Load the program shown in Figure 9-8 into the Trainer. Enter the program starting address into the program counter and single-step through the program. Record the specified information after each step.

Step 1 display ----- ACCA ---

Step 2 display ----- ACCA ---

Step 3 display ----- ACCA ---

Step 4 display ----- ACCA ---

22. According to the microprocessor, the product of  $4_{16}$  times  $4_{16}$  is  $-_{16}$ .
23. Now that you are becoming acquainted with the instructions described in Unit 2, examine the three instructions introduced in this Experiment. Enter the program listed in Figure 9-9.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	4F	CLRA	Clear accumulator.
0001	97	STA	Store the contents
0002	0A	$0A_{16}$	at this address.
0003	4C	INCA	Increment accumulator.
0004	97	STA	Store the contents
0005	0B	$0B_{16}$	at this address.
0006	4A	DECA	Decrement accumulator.
0007	97	STA	Store the contents
0008	0C	$0C_{16}$	at this address.
0009	3E	HLT	Stop.
000A	FF	$FF_{16}$	Reserved for data.
000B	FF	$FF_{16}$	Reserved for data.
000C	FF	$FF_{16}$	Reserved for data.

Implementation of the Clear, Increment, and Decrement instructions.

24. Set the program counter to 0000 and single-step through the program. Record the specified information after each step.

Step 1 display \_ \_ \_ \_ \_ ACCA \_ \_.

Step 2 display \_ \_ \_ \_ \_ ACCA \_ \_.

Step 3 display \_ \_ \_ \_ \_ ACCA \_ \_.

Step 4 display \_ \_ \_ \_ \_ ACCA \_ \_.

Step 5 display \_ \_ \_ \_ \_ ACCA \_ \_.

Step 6 display \_ \_ \_ \_ \_

25. Compare your accumulated data with the program in Figure 9-9. Note that when op codes  $4F_{16}$ ,  $4C_{16}$ , and  $4A_{16}$  are executed, the single-step display advances only one address location. This is because of their inherent addressing mode; immediate and direct addressing modes require two locations in memory.
26. Shown below is a program to swap the contents of two memory locations. Now examine the process using the Trainer. Enter the program listed in Figure 9-10.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	96	LDA	Load accumulator direct with operand 1
0001	10	$10_{16}$	stored at this address.
0002	97	STA	Store operand 1
0003	12	$12_{16}$	at this address.
0004	96	LDA	Load accumulator direct with operand 2
0005	11	$11_{16}$	stored at this address.
0006	97	STA	Store operand 2
0007	10	$10_{16}$	at this address.
0008	96	LDA	Load accumulator direct with operand 1
0009	12	$12_{16}$	stored at this address.
000A	97	STA	Store operand 1
000B	11	$11_{16}$	at this address.
000C	4F	CLRA	Clear the accumulator.
000D	97	STA	Store the contents
000E	12	$12_{16}$	at this address.
000F	3E	HLT	Stop.
0010	AA	$170_{10}$	Operand 1.
0011	BB	$187_{10}$	Operand 2.
0012	00	00	Temporary storage.

Data transfer between two addresses.

27. Set the program counter to starting address 0000 and single-step through the program. Record the specified information after each step.

Step 1 display \_ \_ \_ \_ \_ .      ACCA \_ \_ .

Step 2 display \_ \_ \_ \_ \_ .      ACCA \_ \_ .

Step 3 display \_ \_ \_ \_ \_ .      ACCA \_ \_ .

Step 4 display \_ \_ \_ \_ \_ .      ACCA \_ \_ .

Step 5 display \_ \_ \_ \_ \_ .      ACCA \_ \_ .

Step 6 display \_ \_ \_ \_ \_ .      ACCA \_ \_ .

Step 7 display \_ \_ \_ \_ \_ .      ACCA \_ \_ .

Step 8 display \_ \_ \_ \_ \_ .      ACCA \_ \_ .

28. Examine address:

0010 \_ \_ .

0011 \_ \_ .

0012 \_ \_ .

29. Compare your accumulated data with the program in Figure 9-10.

30. Now you will examine some common programming pitfalls. Without modifying the previous program, except as directed in Figure 9-11, enter the program listed in Figure 9-11.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	86	LDA	Load accumulator immediately with operand 1.
0001	4F	79 <sub>10</sub>	
0002	97	STA	Store operand 1 at this address.
0003	05	05 <sub>16</sub>	
0004	4A	DECA	Decrement accumulator
0005	3E	HLT	Stop.

Figure 9-11

Storing data at an address in the program.

31. Set the program counter to 0000 and single-step through the program. Record the specified information after each step.

Step 1 display	-----	ACCA	---
Step 2 display	-----	ACCA	---
Step 3 display	-----	ACCA	---
Step 4 display	-----	ACCA	---
Step 5 display	-----	ACCA	---
Step 6 display	-----	ACCA	---
Step 7 display	-----	ACCA	---
Step 8 display	-----	ACCA	---
Step 9 display	-----	ACCA	---

32. Compare your accumulated data with the program in Figure 9-11. Note that the data in the accumulator (operand 1) has been stored at address 0005. This removed the HLT instruction and allowed the microprocessor to continue executing any valid instructions in memory. In this case, the remaining unaltered instructions from the previous program are used. When you write a program, **make sure** you do not store data at an address that contains a needed instruction or data.
33. Using the data you accumulated in step 31 of this experiment, plus the programs listed in Figures 9-10 and 9-11, determine the contents of address:

0010 \_ \_.

0011 \_ \_.

0012 \_ \_.

34. Now examine the Trainer contents at address:

0010 \_ \_.

0011 \_ \_.

0012 \_ \_.

Your estimated data from step 33, and the actual contents should be identical. If they are not, re-examine your calculations and the contents of each memory location from 0000 to 0012. You might have inadvertently modified the contents of an address in the previous steps.

35. Without modifying the previous program, except as directed in Figure 9-12, enter the program listed in Figure 9-12.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	86	LDA	Load accumulator immediately with
0001	40	64 <sub>10</sub>	operand 1.
0002	8B	ADD	Add to accumulator immediately with
0003	0A	10 <sub>10</sub>	operand 2.
0004	97	STA	Store the sum
0005	07	07 <sub>16</sub>	at this address.
0006	4F	CLRA	Clear accumulator.
0007	00	00	Reserved for data.

Figure 9-12

Addition of two numbers with immediate addressing.

36. Set the program counter to 0000 and single-step through the program. Record the specified information after each step.

Step 1 display	-----	ACCA	---
Step 2 display	-----	ACCA	---
Step 3 display	-----	ACCA	---
Step 4 display	-----	ACCA	---
Step 5 display	-----	ACCA	---
Step 6 display	-----	ACCA	---
Step 7 display	-----	ACCA	---
Step 8 display	-----	ACCA	---
Step 9 display	-----	ACCA	---

37. Compare your accumulated data with the program in Figure 9-12. Note that the Trainer executed the instructions beyond address 0007. This occurred because there was no halt instruction in the program. Always end your program with a halt instruction. If you don't, the microprocessor will try to execute all of the information contained in memory, thinking it is part of the program. In the process, the program you entered may get modified.
38. This final programming pitfall illustrates a problem almost everybody experiences. Enter the program listed in Figure 9-13.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	96	LDA	Load accumulator direct with
0001	07	07 <sub>16</sub>	operand 1 stored at this address.
0002	8B	ADD	Add to accumulator direct with
0003	07	07 <sub>16</sub>	operand 1 stored at this address.
0004	8B	ADD	Add to accumulator direct with
0005	07	07 <sub>16</sub>	operand 1 stored at this address.
0006	3E	HLT	Stop.
0007	05	05 <sub>10</sub>	Operand 1.

Figure 9-13

Multiplication of two numbers using successive addition in the direct addressing mode.

39. Set the program counter to 0000 and single-step through the program. Record the specified information after each step.

Step 1 display \_ \_ \_ \_ \_ ACCA \_ \_.

Step 2 display \_ \_ \_ \_ \_ ACCA \_ \_.

Step 3 display \_ \_ \_ \_ \_ ACCA \_ \_.



40. Compare your accumulated data with the program in Figure 9-13. The program should have added 05 three times ( $5 \times 3$ ) for the answer 0F. The Trainer indicates the answer is 13. This discrepancy occurred because the program contains the wrong **addressing mode op code** for the ADD function. It should be 9B rather than 8B. Return to Figure 9-13 and change the two ADD op codes to 9B so the program will be correct.
41. In Unit 2, you were shown that RAM (random access memory) was a read/write type memory, while ROM (read only memory) is a preprogrammed memory that can only be read and not written into. To examine these memory types, enter FF at address 0000 through 000F.
42. Examine the following memory locations and write down the contents next to each address. Use the first data column for each address. You will use the second column later.

ADDRESS	DATA	DATA	ADDRESS	DATA	DATA
0000	--	--	FD00	--	--
0001	--	--	FD01	--	--
0002	--	--	FD02	--	--
0003	--	--	FD03	--	--
0004	--	--	FD04	--	--
0005	--	--	FD05	--	--
0006	--	--	FD06	--	--
0007	--	--	FD07	--	--
0008	--	--	FD08	--	--
0009	--	--	FD09	--	--
000A	--	--	FD0A	--	--
000B	--	--	FD0B	--	--
000C	--	--	FD0C	--	--
000D	--	--	FD0D	--	--
000E	--	--	FD0E	--	--
000F	--	--	FD0F	--	--

43. Turn the Trainer power off, then unplug the line cord. Wait twenty seconds, then plug in the line cord and turn on the Trainer.

44. Examine the memory locations listed in step 42, and write down the contents next to each address, in the second data column. Compare the two sets of data. Notice the data obtained at address 0000 through 000F changed when all Trainer power was removed. However, the data at address FD00 through FD0F is unchanged. Address 0000 is RAM, while address FD00 is ROM. Memory is lost from RAM when power is removed. When power is reapplied, random data will appear in the memory.

Enter FF at address FD00 through FD0F. Now examine address FD00 through FD0F. Notice the data is identical to that obtained in step 42. This shows that ROM can not be written into. You can send data down the data bus, but the memory will not accept it.

SUGGESTION: Use the nine instructions presented and write a few sample programs of your own. It's quite simple and can be great fun.

## Experiment 4

### ARITHMETIC AND LOGIC INSTRUCTIONS

#### OBJECTIVES:

To present seven new instructions and use them in simple programs.

To demonstrate 2's complement conversion.

To demonstrate binary subtraction.

To demonstrate binary addition of signed numbers.

To demonstrate logical manipulation of data using the AND and OR instructions.

#### Introduction

In Experiment 3, you used nine instructions to write various programs. These instructions were:

MNEMONIC	OP CODE	ADDRESSING MODE
LDA	86 <sub>16</sub>	Immediate
LDA	96 <sub>16</sub>	Direct
ADD	8B <sub>16</sub>	Immediate
ADD	9B <sub>16</sub>	Direct
STA	97 <sub>16</sub>	Direct
CLRA	4F <sub>16</sub>	Inherent
INCA	4C <sub>16</sub>	Inherent
DECA	4A <sub>16</sub>	Inherent
HLT	3E <sub>16</sub>	Inherent

Seven new instructions are presented in this experiment. Each is listed in Figure 9-14.

Unit 3 examined the process of binary arithmetic, 2's complement arithmetic, signed number addition, and Boolean logic. Through sample programs, this experiment will illustrate some of the operations presented in Unit 3.

NAME	MNEMONIC	OPCODE	DESCRIPTION
Complement 2's or Negate (Inherent)	NEGA	0100 0000 <sub>2</sub> or 40 <sub>16</sub>	Replace the contents of the accumulator with its complement plus 1.
Subtract (Immediate)	SUB	1000 0000 <sub>2</sub> or 80 <sub>16</sub>	Subtract the contents of the next memory location from the contents of the accumulator. Place the difference in the accumulator.
Subtract (Direct)	SUB	1001 0000 <sub>2</sub> or 90 <sub>16</sub>	Subtract the contents of the memory location whose address is given by the next byte from the present contents of the accumulator. Place the difference in the accumulator.
AND (Immediate)	ANDA	1000 0100 <sub>2</sub> or 84 <sub>16</sub>	Perform the logical AND between the contents of the accumulator and the contents of the next memory location. Place the result in the accumulator.
AND (Direct)	ANDA	1001 0100 <sub>2</sub> or 94 <sub>16</sub>	Perform the logical AND between the contents of the accumulator and the contents of the memory location whose address is given by the next byte. Place the result in the accumulator.
OR, Inclusive (Immediate)	ORA	1000 1010 <sub>2</sub> or 8A <sub>16</sub>	Perform the logical OR between the contents of the accumulator and the contents of the next memory location. Place the result in the accumulator.
OR, Inclusive (Direct)	ORA	1001 1010 <sub>2</sub> or 9A <sub>16</sub>	Perform the logical OR between the contents of the accumulator and the contents of the memory location whose address is given by the next byte. Place the result in the accumulator.

Figure 9-14

Instructions introduced in this experiment.

## Procedure

1. In the first part of the experiment, you will determine how the microprocessor represents negative and positive numbers. The program shown in Figure 9-15 loads a positive number into the accumulator and then repeatedly decrements the number until it is negative. Enter this program into the Trainer. Verify that you entered it properly by examining each address.
2. Go to the single-step mode by: pressing the PC key; pressing the CHAN key; and entering the starting address (0000). Single-step through the program by repeatedly pressing the SS key. Notice that the first instruction places  $+5_{10}$  in the accumulator. Refer to Figure 9-16 and record the contents of the accumulator (in both hexadecimal and binary) after each DECA instruction is executed.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	86	LDA	Load accumulator immediate with 05.
0001	05	05	
0002	4A	DECA	Repeatedly decrement the accumulator.
0003	4A	DECA	
0004	4A	DECA	
0005	4A	DECA	
0006	4A	DECA	
0007	4A	DECA	
0008	4A	DECA	
0009	4A	DECA	
000A	4A	DECA	
000B	4A	DECA	
000C	4A	DECA	
000D	4A	DECA	
000E	4A	DECA	Halt
000F	3E	HLT	

Figure 9-15

This program decrements the contents of the accumulator from +5 to -8.

AFTER STEP	CONTENTS OF ACCUMULATOR		
	DECIMAL	HEXADECIMAL	BINARY
1	+5	05	0000 0101
2	+4		
3	+3		
4	+2		
5	+1		
6	0		
7	-1		
8	-2		
9	-3		
10	-4		
11	-5	FB	1111 1011

Figure 9-16

Record results here.

- In step 7, the number in the accumulator changed from 0 to -1. The microprocessor expresses -1 as  $_{-16}$  or  $_{-2}$ . The table you have developed in Figure 9-16 shows how the microprocessor expresses the signed number from +5 to -5 in both hexadecimal and binary. The next program will add signed numbers like these.
- Enter the program shown in Figure 9-17. Use the single step mode to execute the program. What number is in the accumulator after the first instruction is executed?  $_{-16}$  or  $_{-2}$ . What signed decimal number does this represent? \_\_\_\_\_.
- What number is in the accumulator after the second instruction is executed?  $_{-16}$  or  $_{-2}$ . What decimal number does this represent? \_\_\_\_\_.
- What number is in the accumulator after the third instruction is executed?  $_{-16}$  or  $_{-2}$ . What signed decimal number does this represent? \_\_\_\_\_.

## Discussion

These very simple examples illustrate how the microprocessor represents signed numbers. Further experiments will show that the microprocessor can represent signed numbers between  $+127_{10}$  and  $-128_{10}$ . You could determine the bit pattern for each negative number by clearing the accumulator and decrementing the required number of times. However, there are much simpler ways of determining the proper bit pattern for negative numbers.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	86	LDA	Load accumulator immediate
0001	05	+5	with +5.
0002	8B	ADD	Add immediate
0003	FB	-5	-5.
0004	8B	ADD	Add immediate
0005	FC	-4	-4
0006	3E	HLT	

Figure 9-17  
Adding signed numbers.

The simplest way is to start with the positive binary equivalent and take the two's complement by changing all 0's to 1's and 1's to 0's and adding 1. The microprocessor has an instruction that will do this for us. It is called the two's complement or Negate instruction. Its mnemonic is NEGA. This instruction changes the number in the accumulator to its two's complement. It is used to change the sign of a number.

## Procedure (Continued)

7. Load the program shown in Figure 9-18. Use the single-step mode to execute the program. Execute the first instruction by depressing the SS key. What number is in the accumulator?  $_{-16}$  or  $_{-2}$ . What signed decimal number does this represent? \_\_\_\_\_.
8. Execute the second instruction. What number is in the accumulator?  $_{-16}$  or  $_{-2}$ . What signed decimal number does this represent? \_\_\_\_\_. Compare this with the number in step 7. What affect did the NEGA instruction have? \_\_\_\_\_.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	86	LDA	Load accumulator immediate
0001	05	+5	with +5.
0002	40	NEGA	Change the number to -5.
0003	40	NEGA	Change it back to +5.
0004	4A	DECA	Decrement the number to +4.
0005	40	NEGA	Change the number to -4.
0006	40	NEGA	Change it back to +4.
0007	3E	HLT	Halt

Figure 9-18  
Using the NEGA instruction.

9. Execute the third instruction. What number is in the accumulator?  
—  $-_{16}$  or — — — — —  $-_2$ . What signed decimal number does this represent? \_\_\_\_\_. Is your answer the same as that found in step 7? \_\_\_\_\_.
10. Execute the fourth instruction. This decrements the accumulator so that it now contains the signed decimal number \_\_\_\_\_.
11. Execute the fifth instruction. What number is in the accumulator?  
—  $-_{16}$  or — — — — —  $-_2$ . What signed decimal number does this represent? \_\_\_\_\_.
12. Execute the sixth instruction. The number in the accumulator is  
—  $-_{16}$  once more.

## Discussion

The program used the NEGA instruction four times. The first time, the NEGA instruction changed  $05_{16}$  to its two's complement  $FB_{16}$ . Referring back to the table you developed in Figure 9-16, this is the representation for  $-5_{10}$ . Thus, the NEGA instruction effectively changes the sign of the number in the accumulator. The next step proved this again by converting  $-5_{10}$  back to  $+5_{10}$ . To further emphasize the point, the number was decremented to  $+4_{10}$ . The next NEGA instruction changed this to  $FC_{16}$  which is the representation for  $-4_{10}$ . The final NEGA instruction converts this back to  $+4_{10}$ . This instruction allows us to convert a positive number to its negative equivalent and vice versa.

In Unit 3, you learned that the MPU can work with signed numbers in the range of  $+127_{10}$  to  $-128_{10}$  or unsigned numbers in the range of 0 to  $255_{10}$ . This capability results from the way we interpret bit patterns. The following steps will demonstrate this.

## Procedure (Continued)

13. Figure 9-19 shows a program for adding the unsigned numbers  $220_{10}$  and  $27_{10}$ . Load this program into the Trainer and execute it. The final result in the accumulator is —  $-_{16}$  or — — — — —  $-_2$ . What unsigned decimal number does this represent? \_\_\_\_\_.



HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	86	LDA	Load accumulator immediate
0001	DC	220 <sub>10</sub>	with 220 <sub>10</sub> .
0002	8B	ADD	Add immediate
0003	1B	27 <sub>10</sub>	27 <sub>10</sub> .
0004	3E	HLT	Halt.

Figure 9-19

Adding unsigned numbers.

14. Figure 9-20 shows a program for adding the signed numbers  $-36_{10}$  and  $27_{10}$ . Load and execute this program. The final result in the accumulator is  $_{-16}$  or  $_{-2}$ . What signed decimal number does this represent? \_\_\_\_\_.
15. Compare the results obtained in steps 13 and 14. Compare the HEX Contents columns of Figure 9-19 with that of Figure 9-20.

## Discussion

This demonstrates that the MPU simply adds bit patterns. It is our interpretation of these patterns that decide whether we are using signed or unsigned numbers. After all, the two programs are identical except for our interpretation of the input and output data.

Negative numbers are often encountered when performing subtract operations. The subtract instruction was shown earlier in Figure 9-14. Either immediate or direct addressing can be used.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	86	LDA	Load accumulator immediate
0001	DC	$-36_{10}$	with $-36_{10}$
0002	8B	ADD	Add immediate
0003	1B	$+27_{10}$	$+27_{10}$
0004	3E	HLT	Halt.

Figure 9-20

Adding signed numbers.

## Procedure (Continued)

16. Load the program shown in Figure 9-21. Execute the program using the single-step mode. What is the number in the accumulator after the first subtract instruction is executed?  $-16_{10}$  or  $-----2$  or  $-10_{10}$ .
17. What is the number in the accumulator after the second subtract instruction is executed?  $-----16$  or  $-----2$ . What signed decimal number does this represent? \_\_\_\_\_.

## Discussion

The first subtract instruction subtracted  $16_{10}$  from  $47_{10}$ , leaving  $31_{10}$ . The second one subtracted  $35_{10}$  from  $31_{10}$ . This produced a result of  $-4_{10}$ . However, the MPU expressed  $-4$  in two's complement form ( $FC_{16}$  or  $1111\ 1100_2$ ). You will find this to be the case anytime the MPU produces a negative result.

Now let's look at some of the logical instructions available to the microprocessor. The AND and OR instructions are described in Figure 9-14. Carefully read the description of these instructions given there. While these instructions have many uses, we will demonstrate only one here. Earlier you learned that certain peripheral devices communicate with computers using the ASCII code. Thus, when the "2" key on a teletypewriter is pushed, the computer receives the ASCII code for 2, which is  $0011\ 0010$ . The ASCII code for 6 is  $0011\ 0110$ . Notice that the four least significant bits of the ASCII character are the binary value of the corresponding numeral. Thus, we can convert the ASCII characters for the numerals 0 through 9 to binary simply by setting the four most significant bits to 0's. Likewise, we can convert the binary numbers  $0000\ 0000$  through  $0000\ 1001$  to ASCII by changing the four most significant bits to  $0011$ .

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	86	LDA	Load accumulator immediate
0001	2F	$47_{10}$	with $47_{10}$ .
0002	80	SUB	Subtract immediate
0003	10	$16_{10}$	$16_{10}$
0004	80	SUB	Subtract immediate
0005	23	$35_{10}$	$35_{10}$
0006	3E	HLT	Halt

Figure 9-21

Using the subtract instruction.

## Procedure (Continued)

18. Load the program shown in Figure 9-22. Single-step through the first instruction. The number in the accumulator is \_\_\_\_\_<sub>2</sub>.
19. Execute the second instruction. This AND's the contents of the accumulator with the "mask" \_\_\_\_\_. The number in the accumulator after this AND operation is \_\_\_\_\_<sub>2</sub>. Compare this with the number that was in the accumulator in step 18. Compare both numbers with the mask. A 1 in the original number is retained only if there is a \_\_\_\_\_ in the corresponding bit position of the mask.
20. Execute the third instruction. In what memory location is the number in the accumulator stored? \_\_\_\_\_<sub>16</sub>. What number is now in the accumulator? \_\_\_\_\_<sub>2</sub>. Does the number still appear in the accumulator after being stored in memory? \_\_\_\_\_.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	96	LDA	Load the accumulator with
0001	0B	OB	the ASCII character at this address.
0002	84	AND	AND it with
0003	0F	OF	this "mask".
0004	97	STA	Store the binary equivalent
0005	0C	OC	at this address.
0006	8A	ORA	OR the number with
0007	30	30	this "mask".
0008	97	STA	Store the result
0009	0D	OD	here.
000A	3E	HLT	Stop
000B	37	0011 0111	ASCII character for numeral 7.
000C	—	—	Reserved
000D	—	—	Reserved

Figure 9-22  
Using the AND and OR instruction.

21. Execute the fourth instruction. This OR's the contents of the accumulator with the "mask"  $\_\_\_\_\_\_2$ . The number in the accumulator is  $\_\_\_\_\_\_2$ . Compare this with the mask and the number that was in the accumulator in step 20. A 1 is produced in the result whenever there is a  $\_\_\_\_\_\_$  in the corresponding bit position of either the original number, the mask, or both.
22. Execute the fifth instruction. This stores the number in memory location  $\_\_\_\_\_\_{16}$ .
23. Examine memory locations  $000B_{16}$ ,  $000C_{16}$ , and  $000D_{16}$  and compare their contents.

## Discussion

The program first converts the ASCII code for the number "7" to the binary number  $0000\ 0111$ . It does this by ANDing the ASCII code with the "mask"  $0000\ 1111_2$ . Notice that a 1 bit in the mask allows the corresponding bit in the original number to be retained. The four most significant bits of the original number are "masked off" because they are ANDed with 0's.

The OR operation restores the ASCII character by attaching  $0011$  as the four most significant bits.

## Experiment 5

### PROGRAM BRANCHES

#### OBJECTIVES:

To manipulate the N, Z, V, and C condition code registers and determine the conditions that set and reset these flags.

To verify the operation of a simple multiply by repeated addition program that uses the BEQ conditional branch instruction and the BRA instruction.

To demonstrate the ability to write a program that divides by repeated subtraction and uses a conditional branch and BRA instruction.

To introduce a shorthand method of calculating relative addresses.

To verify the operation of a program that converts BCD numbers to their binary equivalent.

To demonstrate the effect an incorrect relative address can have on a program operation and how the microprocessor trainer can be used to debug programs.

#### Introduction

As mentioned previously, conditional branch instructions give the computer the power to make decisions. As the name implies, a certain condition must be met before a branch takes place. The condition code registers monitor the accumulator and signal the presence of a specific condition. If the MPU encounters a conditional branch instruction, it merely checks the condition code registers, or flags, to see if the condition is satisfied. If the specific flag is set, the program branches off to another section. If not, the normal program continues.

Therefore, the conditional branch instructions inherit their power from these simple condition code registers. A sound knowledge of how these flags are set and cleared will enhance your ability as a programmer.



Figure 9-23

Displaying the conditions of the flags.

Since condition code registers are very important, your Trainer was designed with a special key to allow you to examine these flags. The key is labelled "CC" for "Condition Code." When this key is pressed, the state of the condition code registers will be displayed. Each LED displays the contents of one register. The letter just to the right of each LED denotes the corresponding register as shown in Figure 9-23.

Notice that there are six flag registers. For the moment we aren't concerned with the two left-most flags. They will be covered in a later unit. However, we are interested in the N, Z, V, and C flags, because they indicate conditions that can lead to conditional branches. Notice that the flags can either be set as indicated by a 1 or they can be cleared as indicated by a 0.

In this first portion of the experiment, you will implement a "do-nothing" program that manipulates the condition code registers. Then single-stepping through the program, you will examine how the accumulator changes these flags.

## Procedure

1. Turn on the Trainer and then press the RESET key.
2. Now, load the program listed in Figure 9-24 into the Trainer. Once the program is loaded, go back and examine it to insure that it's entered correctly.

Now look at the first instruction of the program in Figure 9-24. It has the op code 01 and the mnemonic is "NOP." As the comments column points out, this is a "do-nothing" type of instruction called a "No-Op." In other words, it performs no operation. In this program, the NOP's primary function is to allow you to see the first instruction before it's executed.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	01	NOP	"DO Nothing" Instruction
0001	86	LDA	Load the accumulator immediate
0002	FF	FF <sub>16</sub>	with FF <sub>16</sub> .
0003	86	LDA	Load the accumulator immediate
0004	77	77 <sub>16</sub>	with 77 <sub>16</sub> .
0005	86	LDA	Load the accumulator immediate
0006	00	00 <sub>16</sub>	with 00 <sub>16</sub> .
0007	86	LDA	Load the accumulator immediate
0008	01	01 <sub>16</sub>	with 01 <sub>16</sub> .
0009	86	LDA	Load the accumulator immediate
000A	92	92 <sub>16</sub>	with 92 <sub>16</sub> .
000B	8B	ADD	Add Immediate
000C	C6	C6 <sub>16</sub>	C6 <sub>16</sub>
000D	86	LDA	Load the accumulator immediate
000E	08	08 <sub>16</sub>	with 08 <sub>16</sub> .
000F	8B	ADD	Add Immediate
0010	08	08 <sub>16</sub>	08 <sub>16</sub> .
0011	86	LDA	Load the accumulator immediate
0012	01	01 <sub>16</sub>	with 01 <sub>16</sub> .
0013	80	SUB	Subtract immediate
0014	02	02 <sub>16</sub>	02 <sub>16</sub> .
0015	86	LDA	Load the accumulator immediate
0016	77	77 <sub>16</sub>	with 77 <sub>16</sub> .
0017	80	SUB	Subtract immediate
0018	66	66 <sub>16</sub>	66 <sub>16</sub> .
0019	86	LDA	Load the accumulator immediate
001A	49	49 <sub>16</sub>	with 49 <sub>16</sub> .
001B	8B	ADD	Add immediate
001C	60	60 <sub>16</sub>	60 <sub>16</sub> .
001D	86	LDA	Load the accumulator immediate
001E	10	10 <sub>16</sub>	with 10 <sub>16</sub> .
001F	3E	HLT	Halt.

Figure 9-24

Program to illustrate the condition code registers.

In previous experiments, you probably noticed that when you single-stepped through programs, you never saw the first instruction. This is because in the "SS" mode, the Trainer executes the first instruction automatically and then stops on the second instruction. This can be somewhat confusing.

To offset this problem, we merely insert the NOP. The Trainer "sees" this as the first instruction, although nothing is accomplished by the NOP. Therefore, the Trainer displays the next instruction, which is the first "real" instruction of the program, permitting you to view it before it's executed.

3. Load the program counter with address 0000 and then press the SS key. Recall that the first four displays represent the address that's currently in the program counter. The two right-most displays show the op code stored at this address. Record the information below.

PC \_ \_ \_ \_ OP CODE \_ \_

Now, press the ACCA key and record the contents of the accumulator.

ACCA \_ \_

The contents of the accumulator will be a random number, since we haven't yet executed a program instruction.

Now, press the CC key and record the contents of the N, Z, V, and C condition code registers below.

\_ \_ \_ \_

N Z V C

Again, the states of the flags are random at this time.

4. Now, press the SS key and then the ACCA key. Record the contents of the accumulator below.

ACCA \_ \_

Press key CC and record the state of the N flag below.

\_ \_ \_ \_

N Z V C

With the negative number  $FF_{16}$  in the accumulator, the negative (N) flag is set.



5. Press the SS key again. The program count should now be  $0005_{16}$  and the op code at this address is 86. Now check and record the contents of the accumulator and the N flag.

ACCA --    -- -- --  
           N Z V C

With the positive number  $77_{16}$  in the accumulator, the N flag is cleared, or reset, to 0.

From the information gathered in steps 4 and 5, what conclusions do you reach with respect to the N flag and the contents of the accumulator?

---

6. Single-step the program again. The program count is now  $0007_{16}$ . Record the contents of the accumulator and the condition of the Z flag below.

ACCA --    -- -- --  
           N Z V C

With  $00_{16}$  in the accumulator, the Z flag is set.

Press SS and again record the contents of the accumulator and the Z flag below.

ACCA --    -- -- --  
           N Z V C

The accumulator now contains  $01_{16}$  and the Z flag is cleared. What is the relation between the contents of the accumulator and the Z, or zero flag?

---

7. Single-step again and record the information below.

ACCA --      -- = --  
N Z V C

This step loads the number  $92_{16}$  into the accumulator. Bit 7 of the accumulator contains a  $1_2$  so the N flag is set. Naturally, the Z flag is cleared. The next instruction will add  $C6_{16}$  to the contents of the accumulator. As shown below, this operation should generate a carry.

1001	0010	=	$92_{16}$
1100	0110	=	$C6_{16}$
<hr/>			
0101	1000	=	$158_{16}$

CARRY  $\xrightarrow{1}$

Press the SS key and record the information below.

ACCA --      -- = --  
N Z V C

The 8-bit accumulator cannot hold the 9-bit sum. However, the carry generated by the addition sets the C flag.

8. This step loads the number  $08_{16}$  into the accumulator. Press the SS key and record the information below.

ACCA --      -- = --  
N Z V C

Notice that loading this new number into the accumulator didn't affect the carry (C) flag. The next step will add  $08_{16}$  to the contents of the accumulator ( $08_{16}$ ).

9. Press the SS key and record the information below.

ACCA --      -- = --  
N Z V C

The accumulator now contains the sum of the addition ( $10_{16}$ ) and the carry flag is cleared.

From the results of steps 8 and 9, you might conclude that the carry flag can be cleared by another \_\_\_\_\_ that does not result in a carry.

10. Press the SS key. The program count should now be 0013. Record the information below.

ACCA \_ \_ \_ \_  
N Z V C

This shows that the accumulator contains  $01_{16}$  and that the N, Z, and C flags are all cleared. When the next instruction is executed, the number  $02_{16}$  will be subtracted from  $01_{16}$  (the contents of the accumulator). As shown below, the subtraction should result in a borrow, setting the C flag.

$$\begin{array}{rcll} & & & \uparrow 1 \\ \text{Borrow} & \text{---} & & \\ & 0000 & 0001 & = 01_{16} \\ & 0000 & 0010 & = 02_{16} \\ \hline & 1111 & 1111 & = FF_{16} \end{array}$$

Notice that the difference is  $\text{FF}_{16}$ . This will set/clear the N flag.

11. Press the SS key and record the information below.

ACCA \_ \_ \_ \_ \_  
NZVC

As expected, the difference produced is  $FF_{16}$ . Also, the N flag is set, indicating a negative number is in the accumulator and the C flag indicates a borrow occurred.

The next step will execute the instruction that loads  $77_{16}$  into the accumulator. After this LDA operation, the C flag will be           .  
set/cleared

12. Press the SS key and record the information below.

ACCA \_ \_ \_ \_ = \_ \_  
N Z V C

Notice that the C flag is still set and that  $77_{16}$  is in the accumulator. Now we will subtract  $66_{16}$  from the accumulator contents ( $77_{16}$ ).


Press the SS key and record the information below.

ACCA \_ \_ \_ \_ = \_ \_  
N Z V C

The difference ( $11_{16}$ ) is now stored in the accumulator and, since no borrow is generated, the C flag is cleared.

13. In this step, the first instruction loads the accumulator with the number  $49_{16}$ . The next instruction adds the number  $60_{16}$  to  $49_{16}$ . As shown below, the addition of these numbers causes an overflow into the sign bit (bit 7) and the sum,  $A9_{16}$ , appears to be a negative number.

0100	1001	=	$49_{16}$
0110	0000	=	$60_{16}$
<hr/>			
1010	1001	=	$A9_{16}$

Overflow changes  
sign bit. 

Of course, this is incorrect and the MPU must be notified of this overflow. This is the purpose of the V flag.

Press the SS key and record the information below.

ACCA \_ \_ \_ \_ = \_ \_  
N Z V C

The number  $49_{16}$  is in the accumulator and the N, Z, V, and C flags are cleared.

Single-step once more and then record the information below.

ACCA -- -- --  
N Z V C

The sum  $A9_{16}$  is now in the accumulator. Notice that the N and V flags are set, indicating that the number in the accumulator is negative and that an overflow occurred.

14. When the next instruction is executed, the number  $10_{16}$  will be loaded into the accumulator.

Single-step the program and record the information below. Notice that the op code 3E (a halt) is the next instruction, so the program is finished.

ACCA -- -- --  
N Z V C

The accumulator contains the number  $10_{16}$ , and all flags cleared. From this, you might conclude that any instruction that doesn't produce an overflow in the accumulator will            the V flag.  
set/clear

## Discussion

In this portion of the experiment, you stepped through a simple program that manipulated the condition code registers. In step 4, the negative number  $FF_{16}$  was loaded into the accumulator. This set the N flag to 1. In step 5, the positive number  $77_{16}$  was loaded into the accumulator. And, as you noted, the N flag was cleared or reset to 0. From these two steps you should have concluded that when the number in the accumulator is negative, the N flag is set. And when the accumulator contains a positive number, the N flag is cleared.

In step 6, the accumulator was loaded with  $00_{16}$ . This set the Z flag to 1. Next, when  $01_{16}$  was loaded, the Z flag was reset or cleared to 0. Your conclusion should have been that when the accumulator contains  $00_{16}$ , the Z flag is set. If it contains any number other than  $00_{16}$ , the Z flag is cleared.

Next, you examined the C flag. When a carry was generated by the addition of the two numbers,  $92_{16}$  and  $C6_{16}$ , the C flag was set. In step 8, you noted that merely loading a new number into the accumulator did not clear the C flag. The carry flag was cleared by another addition that did not result in a carry. Your conclusion should have been that the C flag can only be cleared by an arithmetic operation that does not result in a carry.

As you proved in steps 10 and 11, a subtraction that results in a borrow also sets the C flag. Again, the C flag was cleared by an arithmetic operation, in this case a subtraction, that did not generate a borrow. Therefore, the C flag can only be cleared or reset to  $0_2$  by an arithmetic operation that does not result in a borrow or carry.

You concluded this phase of the experiment by adding two positive numbers, the sum of which overflowed into the sign bit of the accumulator. This set the V or overflow flag, showing that the sum should not be a negative number as the N flag indicated. The next LDA instruction cleared the V flag. From this, you should conclude that the V flag is cleared by any instruction that doesn't produce an overflow.

In the next sections of this experiment, you will step through a few branching programs that illustrate the use of the branch always (BRA) instruction and certain conditional branch instructions. These branch instructions were discussed in Unit 4, and you will verify their operation. We'll begin with the multiply by repeated addition program.

## Procedure (Continued)

15. Enter the program listed in Figure 9-25 into the Trainer. This program multiplies  $05_{16}$  and  $02_{16}$  and stores the product in address  $0013_{16}$ . Recheck the program to insure that it's entered correctly.
16. This is the same program that you stepped through in Unit 4. Notice that the program contains two branch instructions; the BEQ (Branch if Equal Zero) at address  $0005_{16}$  and the BRA (Branch Always) at address  $000E_{16}$ .

The branch if equal zero (BEQ) instruction implies by it's name that a conditional branch will occur when the \_\_\_\_\_ flag is set.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000		CLRA	Clear the accumulator.
0001		→ STA	Store the product
0002		13	in location $13_{16}$ .
0003		LDA	Load the accumulator with the
0004		12	multiplier from location $12_{16}$ .
0005		BEQ	If the multiplier is equal to zero,
0006		09	branch down to the Halt instruction.
0007		DECA	Otherwise, decrement the multiplier.
0008		STA	Store the new value of the
0009		12	multiplier back in location $12_{16}$ .
000A		LDA	Load the accumulator with the
000B		13	product from location $13_{16}$ .
000C		ADD	Add
000D		11	the multiplicand to the product.
000E		BRA	Branch back to instruction
000F		F1	in location 01.
0010		→ HLT	Halt.
0011		05	Multiplicand.
0012		02	Multiplier.
0013		—	Product.

Program to multiply by repeated addition.

17. Now, set the program counter to 0000 and single-step through the program, recording the information in the chart of Figure 9-26. Notice that you will be monitoring the Z flag. A comments column is provided so you can make notes about each step. Use the program listing as a reference for each op code and the corresponding operand.

18. When the BEQ instruction is executed and the Z flag is set, the program branches to the \_\_\_\_\_ instruction.

When the multiplier was  $02_{16}$ , the program halted on the \_\_\_\_\_ pass through the program.

If the multiplier is changed to  $06_{16}$ , how many passes would the program make before it halts? \_\_\_\_\_.

19. Examine the contents of address  $0013_{16}$  and record below.

0013 \_ \_.

STEP	PROGRAM COUNTER	OPCODE	ACCA	Z FLAG	COMMENTS
1					
2					
3					
4					
5					
6					
7					
8					
9					
10					
11					
12					
13					
14					
15					
16					
17					
18					
19					
20					

Figure 9-26

Single-stepping through the Multiply by repeated addition program.



## Discussion

The chart that you completed should be similar to the one shown in Figure 9-27. Compare the charts.

The first step we don't see, since it's executed before the Trainer stops at address 0001. Nevertheless, we do see the result of this clear accumulator instruction because the accumulator contains 00. When step 1 is executed,  $00_{16}$  is stored in location  $0013_{16}$ . Step 2 brings us to address  $0003_{16}$  which loads the accumulator with the multiplier, in this example,  $02_{16}$ . The BEQ instruction is next, but the Z flag is cleared so the program continues on the normal route. Next the multiplier is decremented to  $01_{16}$  and then stored in location  $0012_{16}$ . Now the product ( $00_{16}$ ) is loaded and the multiplicand ( $05_{16}$ ) is added directly. This produces the new product,  $05_{16}$ . Now the program encounters the BRA, or branch always instruction and it branches back to address  $0001_{16}$ .

Here the new product is stored away and the multiplier is loaded again. It's  $01_{16}$  this time, so the program continues on through the BEQ instruction, the multiplier is decremented to  $00_{16}$ , and the multiplicand  $05_{16}$  is added to the product. The new product ( $0A_{16}$ ) is still in the accumulator. Once again, the BRA instruction loops flow back to address  $0001_{16}$  and the product is stored in address  $0013_{16}$ .

The multiplier is now loaded and, since it's been decremented to  $00_{16}$ , it sets the Z flag. The BEQ instruction checks the Z flag, finds that it's set and branches to the halt instruction at address  $0010_{16}$ . Therefore, the program makes two complete passes, before the multiplier becomes  $00_{16}$ . On the third pass through, BEQ terminates the program because the Z flag is set.

The multiplier sets the count and determines how many additions will be performed. If the multiplier is changed to  $06_{16}$ , the program will make six complete loops, halting on the seventh loop. The BEQ will only be satisfied when the multiplier has been reduced to 00.

All branch instructions use relative addressing. In Unit 4, we discussed the method used to calculate the destination address for a branch instruction. However, another shorthand type procedure that's quite popular with programmers can be used. With this technique, you simply count in hexadecimal. For a forward branch, you begin at  $00_{16}$  and count up to the destination address.

STEP	PROGRAM COUNTER	OPCODE	ACCA	Z FLAG	COMMENTS
1	0001	97	00	1	Store the product (00 <sub>16</sub> ) in address 0013 <sub>16</sub> .
2	0003	96	00	1	Load the accumulator with the multiplier (02 <sub>16</sub> ) from address 0012 <sub>16</sub> .
3	0005	27	02 ↑ Multiplier	0	BEQ. Check the Z flag. It's not set so continue.
4	0007	4A	02	0	Decrement the multiplier (02 <sub>16</sub> ).
5	0008	97	01 ↑ New Multiplier	0	Store the new multiplier (01 <sub>16</sub> ) at address 0012 <sub>16</sub> .
6	000A	96	01	0	Load the accumulator with the product (00) at address 0013 <sub>16</sub> .
7	000C	9B	00	1	Add the multiplicand (05) giving new product.
8	000E	20	05 ↑ New Product	0	Branch back to address 0001 <sub>16</sub> .
9	0001	97	05	0	Store the product (05 <sub>16</sub> ) in address 0013 <sub>16</sub> .
10	0003	96	05	0	Load the accumulator with the multiplier (01 <sub>16</sub> ) located at address 0012 <sub>16</sub> .
11	0005	27	01	0	BEQ. Check Z flag. It's not set so continue.
12	0007	4A	01	0	Decrement the multiplier (01 <sub>16</sub> ).
13	0008	97	00 ↑ New Multiplier	1	Store the new Multiplier (00 <sub>16</sub> ) at address 0012 <sub>16</sub> .
14	000A	96	00	1	Load the accumulator with the product (05 <sub>16</sub> ) at address 0013 <sub>16</sub> .
15	000C	9B	05	0	Add the multiplicand (05 <sub>16</sub> ) giving new product.
16	000E	20	0A ↑ New Product	0	Branch back to address 0001 <sub>16</sub> .
17	0001	97	0A	0	Store the product (0A <sub>16</sub> ) in address 0013 <sub>16</sub> .
18	0003	96	0A	0	Load the accumulator with the multiplier (00 <sub>16</sub> ) from address 0012 <sub>16</sub> .
19	0005	27	00	1	BEQ. Check the Z flag. Now it's set. Branch to address 0010 <sub>16</sub> .
20	0010	3E	00	1	Halt.

Figure 9-27

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ HEX CONTENTS
18	20	BRA
19	??	??
1A		
1B		
1C		
1D		
1E		
1F		
20		
21		
22		
23		
24		

Originating address

Destination address

We wish to  
Branch to here

Figure 9-28

For example, in the program of Figure 9-28, we want to branch from address  $18_{16}$  to address  $24_{16}$ . Recall that the relative address is added to the contents of the program counter. After the BRA instruction and its operand (the relative address) have been fetched, the program counter is pointing to address  $1A_{16}$ . Therefore, we begin our count at address  $1A_{16}$ . Then we count forward in hex as shown in Figure 9-29. When we reach the destination address, the hexadecimal count is the relative address. In this case, it's  $0A_{16}$ , and we insert this operand at address  $19_{16}$ .

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ HEX CONTENTS
18	20	BRA
19	0A	0A
00 1A		
01 1B		
02 1C		
03 1D		
04 1E		
05 1F		
06 20		
07 21		
08 22		
09 23		
0A 24		

Originating Address

Destination Address

Relative  
Address

Figure 9-29  
Branching forward

To branch backward in the program, we simply count down using negative hex numbers. It may sound more difficult, but once you are accustomed to it, you will find it easier to use than the previous method you learned.

For example, in the program shown in Figure 9-30A, we wish to branch back to address  $58_{16}$ . The BRA instruction, at address  $5D_{16}$  is fetched and the program count points to address  $5F_{16}$ . Figure 9-30B shows how we calculate the address for this backward branch. We begin with  $FF_{16}$ , and count down. When we reach the destination address ( $58_{16}$ ), the count at that point is the relative address, in this case  $F9_{16}$ .

Figure 9-31 shows another example of computing the relative address for a larger branch. The branch instruction is at address  $B0_{16}$  and therefore, the origination address is  $B2_{16}$ . We calculate the relative address as shown in Figure 9-31B. Starting with  $FF_{16}$  at address  $B1_{16}$  we count down to the destination address  $A0_{16}$ . As the count indicates, the relative address to get to  $A0_{16}$  is  $EE_{16}$ .

HEX ADDRESS		HEX CONTENTS	MNEMONICS/ HEX CONTENTS
56		—	—
57		—	—
58	← Destination Address	—	—
59		—	—
5A		—	—
5B		—	—
5C		—	—
5D		20	BRA
5E	← Originating Address	??	??
5F			

**A**

Program branches to here

HEX ADDRESS		HEX CONTENTS	MNEMONICS/ HEX CONTENTS
56		—	—
57		—	—
F9 58	← Destination Address	—	—
FA 59		—	—
FB 5A		—	—
FC 5B		—	—
FD 5C		—	—
FE 5D		20	BRA
FF 5E	← Originating Address	F9	F9
5F			

**B**

Relative address

Figure 9-30

Branching back

HEX ADDRESS		HEX CONTENTS	MNEMONICS/ HEX CONTENTS
We wish to branch to here	AO	—	—
	A1	—	—
	A2	—	—
	A3	—	—
	A4	—	—
	A5	—	—
	A6	—	—
	A7	—	—
	A8	—	—
	A9	—	—
	AA	—	—
	AB	—	—
	AC	—	—
	AD	—	—
	AE	—	—
	AF	—	—
	BO	26	BNE
	B1	??	??
	B2	—	—

A

HEX ADDRESS		HEX CONTENTS	MNEMONICS/ HEX CONTENTS
Relative Address	EE AO	—	—
	EF A1	—	—
	FO A2	—	—
	F1 A3	—	—
	F2 A4	—	—
	F3 A5	—	—
	F4 A6	—	—
	F5 A7	—	—
	F6 A8	—	—
	F7 A9	—	—
	F8 AA	—	—
	F9 AB	—	—
	FA AC	—	—
	FB AD	—	—
	FC AE	—	—
	FD AF	—	—
	FE BO	26	BNE
	FF B1	EE	EE
	B2	—	—

B

Figure 9-31

In the next section of this experiment, you will write a program that will divide by repeated subtraction. You will probably have two branches in this program; a forward branch and a branch back. Use this new technique to calculate the relative addresses for both branches.

### Procedure (Continued)

20. In Unit 4, we discussed a program that divides by repeated subtraction. The flow chart for this program is shown in Figure 9-32. Using this flow chart as a guide and the instructions presented in Figure 9-33, write a program that divides by repeated subtraction.

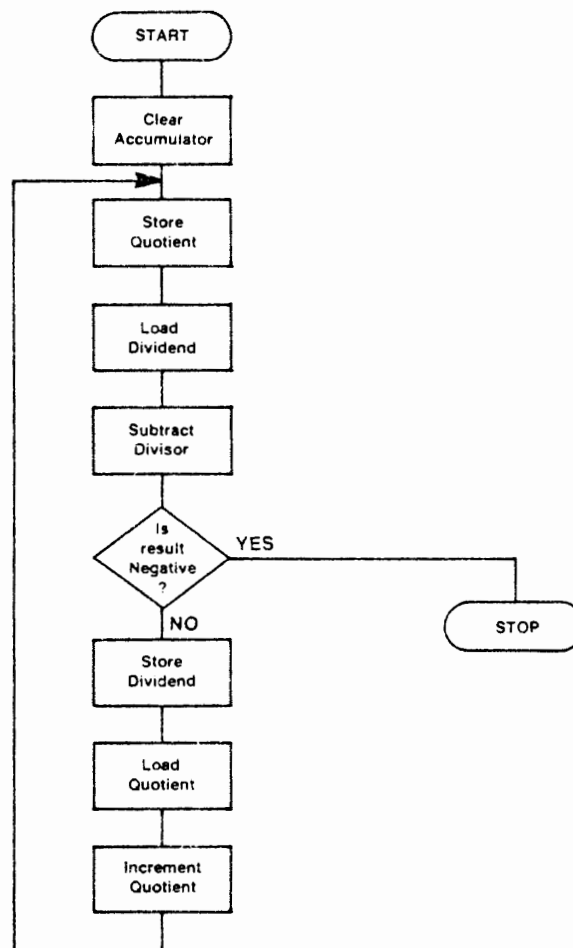


Figure 9-32  
Flow chart for dividing by repeated subtraction.

INSTRUCTION	MNEMONIC	ADDRESSING MODE			
		IMMEDIATE	DIRECT	RELATIVE	INHERENT
Load Accumulator	LDA	86	96		4F
Clear Accumulator	CLRA				4A
Decrement Accumulator	DECA				4C
Increment Accumulator	INCA				
Store Accumulator	STA		97		
Add	ADD	8B	9B		
Subtract	SUB	80	90		
Branch Always	BRA			20	
Branch if Carry Set	BCS			25	
Branch if Equal Zero	BEQ			27	
Branch if Minus	BMI			2B	
Halt	HLT				3E

Figure 9-33

Instructions to be used.

21. Now load the program into the Trainer. Let the dividend be  $0B_{16}$  and the divisor be  $05_{16}$ . Change the program counter to the starting address of your program and single-step through the program, recording the information in the chart of Figure 9-34.
22. Examine the contents of the address that stores the dividend and the quotient. If you followed the flow chart, the address where the dividend is stored should now contain the remainder from the division. Record the contents below.

Quotient \_\_\_\_\_ Remainder \_\_\_\_\_

STEP	PROGRAM COUNTER	OPCODE	ACCA	N FLAG	COMMENTS

Figure 9-34



## Discussion (Continued)

Now you've written a program that incorporates an unconditional branch and a conditional branch. Hopefully, you calculated the relative addresses using the shorthand technique just discussed. Our program for the divide by repeated subtraction is listed in Figure 9-35. If you followed the flow chart, your program should be similar to this.

HEX ADDRESS	HEX CONTENTS	MNEMONIC/HEX CONTENTS	COMMENTS
0000	4F	CLRA	Clear the accumulator.
0001	97	STA	Store in the quotient which
0002	13	13	is at address location 13 <sub>16</sub> .
0003	96	LDA	Load the accumulator with the
0004	11	11	dividend from location 11 <sub>16</sub> .
0005	90	SUB	Subtract the
0006	12	12	divisor from the dividend.
0007	2B	BMI	If the difference is negative,
0008	07	07	branch down to the Halt instruction.
0009	97	STA	Otherwise, store the difference
000A	11	11	back in location 11 <sub>16</sub> .
000B	96	LDA	Load the accumulator with the
000C	13	13	quotient.
000D	4C	INCA	Increment the quotient by one.
000E	20	BRA	Branch back to instruction
000F	F1	F1	in location 01.
0010	3E	HLT	Halt.
0011	0B	0B	Dividend (11 <sub>16</sub> ).
0012	05	05	Divisor (5 <sub>16</sub> ).
0013	—	—	Quotient.

Figure 9-35  
Dividing by repeated subtraction.

STEP	PROGRAM COUNTER	OPCODE	ACCA	N FLAG	COMMENTS
1	0001	97	00	0	Store the quotient (00 <sub>16</sub> ) at address 0013 <sub>16</sub> .
2	0003	96	00	0	Load the accumulator with the dividend from address 0011 <sub>16</sub> .
3	0005	90	0B ↑ Dividend	0	Subtract the divisor (05 <sub>16</sub> ) at address 0012 <sub>16</sub> from the accumulator.
4	0007	2B	06 ↑ After subtraction	0	BMI. Check the N flag. It's not set so continue.
5	0009	97	06	0	Store the difference (06 <sub>16</sub> ) back in address 0011 <sub>16</sub> .
6	000B	96	06	0	Load the accumulator with the quotient (00 <sub>16</sub> ) at address 0013 <sub>16</sub> .
7	000D	4C	00	0	Increment the quotient.
8	000E	20	01 ↑ Quotient after INC	0	Branch back to the instruction at address 0001 <sub>16</sub> .
9	0001	97	01	0	Store the quotient (01 <sub>16</sub> ) at address 0013 <sub>16</sub> .
10	0003	96	01	0	Load the accumulator with the dividend (06 <sub>16</sub> ) at address 0011 <sub>16</sub> .
11	0005	90	06 ↑ Dividend Now	0	Subtract the divisor (05 <sub>16</sub> ) at address 0012 <sub>16</sub> from the accumulator.
12	0007	2B	01 ↑ After Subtraction	0	BMI. Check the N flag. It's not set so continue.
13	0009	97	01	0	Store the difference (01 <sub>16</sub> ) back in address 0011 <sub>16</sub> .
14	000B	96	01	0	Load the accumulator with the quotient (01 <sub>16</sub> ) at address 0013 <sub>16</sub> .
15	000D	4C	01	0	Increment the quotient.
16	000E	20	02 ↑ Quotient after INC.	0	Branch back to the instruction at address 0001 <sub>16</sub> .
17	0001	97	02	0	Store the quotient (02 <sub>16</sub> ) at address 0013 <sub>16</sub> .
18	0002	96	02	0	Load the accumulator with the dividend (01 <sub>16</sub> ) at address 0011 <sub>16</sub> .
19	0005	90	01	0	Subtract the divisor (05 <sub>16</sub> ) at address 0012 <sub>16</sub> from the accumulator.
20	0007	2B	FC ↑ Negative Number	1	BMI. Check the N flag. Now it's set so branch to the instruction at address 0010 <sub>16</sub> .
21	0010	3E	FC	1	Halt.

Figure 9-36

Notice that we used the BMI (Branch if Minus) conditional branch instruction. Therefore, the N or negative flag will satisfy the branch when it's set. Figure 9-36 charts our program as we single-stepped through it. Since the program subtracts the divisor from the dividend and stores the difference as the new dividend, at the conclusion of the program the dividend is actually the remainder of the division. When  $0B_{16}$  is divided by  $05_{16}$ , the quotient should be  $02_{16}$  and the remainder  $01_{16}$ .

So far, we've used the conditional branch instructions only to exit a loop and then halt program execution. However, these branch instructions become even more powerful when they are used to "chain" together different portions of a program. Figure 9-37 shows an example of this chaining effect. The program starts and runs through the first loop until the conditional branch BEQ is satisfied. Then it exits this loop and starts another. When the BEQ condition is satisfied in the second loop, another exit is performed, and another portion of the program is executed.

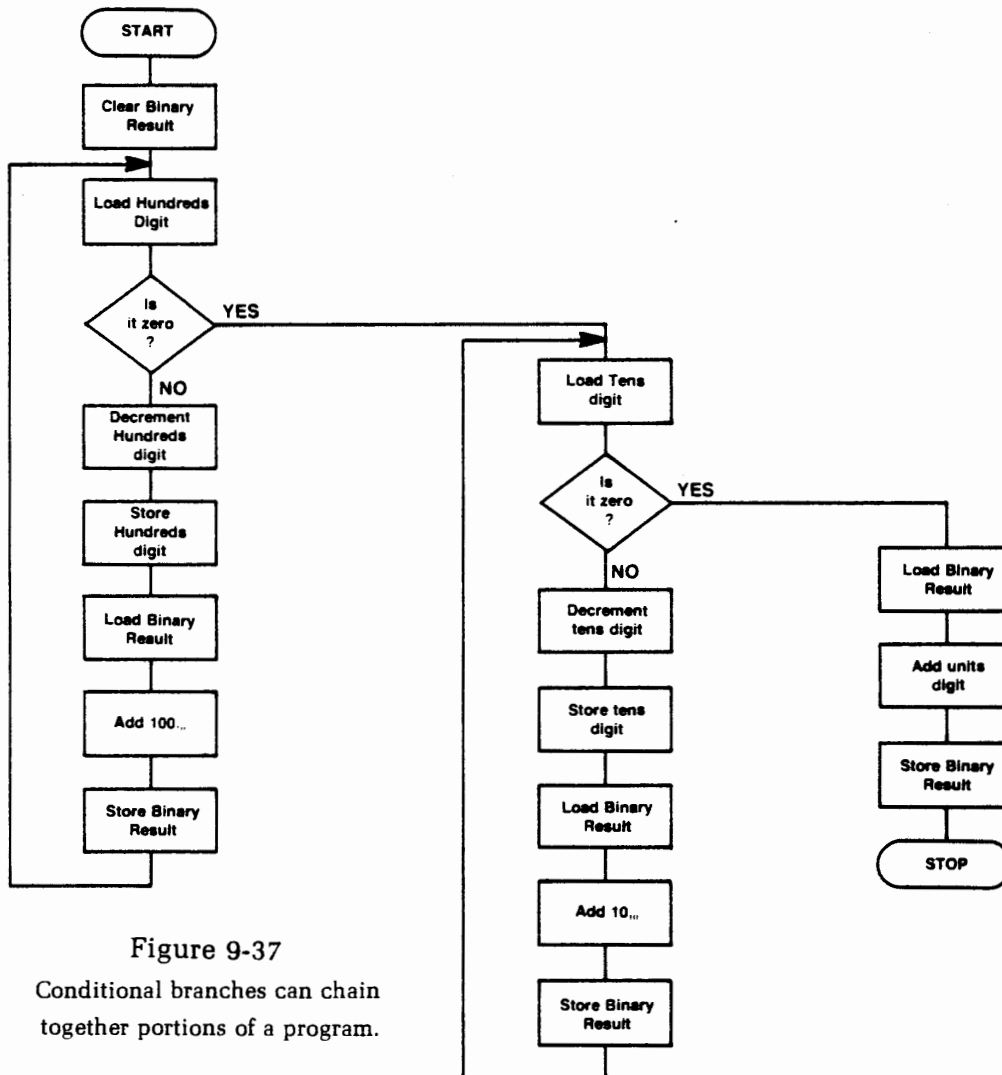


Figure 9-37

Conditional branches can chain together portions of a program.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	4F	CLRA	Clear the Accumulator.
0001	97	STA	Store 00
0002	2B	2B	in location 2B. This clears the binary result.
0003	96	LDA	Load direct into the accumulator
0004	28	28	the hundreds BCD digit.
0005	27	BEQ	If the hundreds digit is zero, branch
0006	0B	0B	forward to the instruction in location 12 <sub>16</sub> .
0007	4A	DECA	Otherwise, decrement the accumulator.
0008	97	STA	Store the result as the new
0009	28	28	hundreds BCD digit.
000A	96	LDA	Load direct into the accumulator
000B	2B	2B	the binary result.
000C	8B	ADD	Add immediate
000D	64	64	100 <sub>10</sub> to the binary result.
000E	97	STA	Store away the new
000F	2B	2B	binary result.
0010	20	BRA	Branch
0011	F1	F1	back to the instruction in location 03 <sub>16</sub> .
0012	96	LDA	Load direct into the accumulator
0013	29	29	the tens BCD digit.
0014	27	BEQ	If the tens BCD digit is zero, branch
0015	0B	0B	forward to the instruction in location 21 <sub>16</sub> .
0016	4A	DECA	Otherwise, decrement the accumulator.
0017	97	STA	Store the result as the new
0018	29	29	tens BCD digit.
0019	96	LDA	Load direct into the accumulator
001A	2B	2B	the binary result.
001B	8B	ADD	Add immediate
001C	0A	0A	10 <sub>10</sub> to the binary result.
001D	97	STA	Store away the new
001E	2B	2B	binary result.
001F	20	BRA	Branch
0020	F1	F1	back to the instruction in location 12 <sub>16</sub> .
0021	96	LDA	Load direct into the accumulator
0022	2B	2B	the binary result.
0023	9B	ADD	Add direct
0024	2A	2A	the units BCD digit.
0025	97	STA	Store away the new
0026	2B	2B	binary result.
0027	3E	HLT	Halt.
0028	01	01	Hundreds BCD digit.
0029	01	01	Tens BCD digit.
002A	07	07	Units BCD digit.
002B	—	—	Reserved for the binary result.

Figure 9-38

Program for converting BCD to binary.

A strategically placed conditional branch at the end of the program can cause a branch back to the beginning that will repeat the program again and again. In the next portion of this experiment, you will load the BCD-to-binary conversion program that you studied earlier. Then you will step through the program and watch as the Trainer executes each instruction.

## Procedure (Continued)

23. Load the program listed in Figure 9-38 into the Trainer. The BCD number  $117_{10}$  will be converted to binary by this program.

The BEQ instruction is used for the conditional branches in this program. This means that MPU will monitor the \_\_\_\_\_ flag to determine if the condition is set.

24. Now set the program counter to 0000 and single-step through the program recording the information in the chart of Figure 9-39. Notice that, at strategic steps, you should stop and answer questions before you continue.

25. What is the hundreds BCD digit at this time? \_\_\_\_\_. The result is now  $64_{16}$ , which is \_\_\_\_\_ in the decimal number system.

Now return to the Trainer and continue stepping through the program.

26. What is the tens BCD digit at this time? \_\_\_\_\_.

The result is now  $6E_{16}$ . This is the equivalent of \_\_\_\_\_ in the decimal number system.

Now return to the Trainer and step through the remainder of the program.

27. Examine address  $002B_{16}$  and record the result below.

\_\_\_\_\_ <sub>16</sub>

Convert this number to its decimal equivalent.

$75_{16} = \text{_____}_{10}$

STEP	PROGRAM COUNTER	OPCODE	ACCA	Z FLAG	COMMENTS
1					
2					
3					
4					
5					
6					
7					
8					
Stop! Return to Step 25.					
9					
10					
11					
12					
13					
14					
15					
16					
17					
18					
Stop! Return to step 26.					
19					
20					
21					
22					
23					
24					
25					

Figure 9-39

## Discussion

Now you've verified the operation of the BCD-to-binary conversion program. The chart that you completed should match the one shown in Figure 9-40.

Since the BEQ instruction is used for the conditional branches in the program, we monitored the Z flag. In this example, the BCD number  $117_{10}$  was converted to its binary equivalent  $75_{16}$ . This program will convert BCD numbers as high as  $255_{10}$ , to their binary equivalent.

The program isn't as complicated as it might appear. The hundreds and tens BCD digits are used to set a count. Each pass through a loop decrements the BCD digit, or count, and then adds the equivalent hexadecimal positional value for that BCD digit. For example, in the hundreds conversion loop,  $64_{16}$  is added to the binary result for each hundreds BCD digit. Hence, the BCD digit sets the count. Then the count is decremented by one and the program loops back and runs through again. When the count is zero, that BCD digit has been added the correct number of times and the program branches off to another loop. This continues until the program halts.

Stepping through the program, you found that after Step 8, the Trainer had completed one loop through the hundreds BCD portion of the program. The count was  $00_{16}$  and the binary result was  $64_{16}$ , or the binary equivalent of  $100_{10}$ . On the next pass through, the program branches to the tens BCD loop.

The first loop through, the tens BCD portion of the program was completed at step 18. The binary result was  $6E_{16}$ , which is the equivalent of  $110_{10}$ . The tens BCD digit had been decremented to  $00_{16}$ . Then all that remained was to add the units BCD digit ( $07_{10}$ ) and the conversion process was complete.

You verified the final result by checking the binary result at location  $002B_{16}$ . Here you found the hex number  $75_{16}$ . When you converted this number to its decimal equivalent, you found that  $75_{16}$  equals  $117_{10}$ . Also, if you converted  $75_{16}$  to binary, you would find the number  $0111\ 0101_2$ , which is the (binary) equivalent of  $117_{10}$ , so the program works.

STEP	PROGRAM COUNTER	OPCODE	ACCA	Z FLAG	COMMENTS
1	0001	97	00	1	Store 00 in address 002B <sub>16</sub> . This clears the binary result.
2	0003	96	00	1	Load the accumulator with the Hundreds BCD digit (01 <sub>16</sub> ).
3	0005	27	Hundreds BCD→ Digit 01	0	BEQ. Check the Z flag. It's clear so continue.
4	0007	4A	01	0	Decrement the BCD Hundreds Digit.
5	0008	97	New→ Hundreds Digit 00	1	Store the new Hundreds Digit (00).
6	000A	96	00	1	Load the accumulator with the Binary Result (00 <sub>16</sub> ).
7	000C	8B	00	1	Add to the binary result 64 <sub>16</sub> .
8	000E	97	Binary→ Result Now 64	0	Store away the new binary result.
9	0010	20	64	0	Branch back to address 0003 <sub>16</sub> .
10	0003	96	64	0	Load the accumulator with the Hundreds BCD digit (00).
11	0005	27	00	1	BEQ. Check the Z flag. It's set so branch to address 0012 <sub>16</sub> .
12	0012	96	00	1	Load the accumulator with the tens BCD digit (01 <sub>16</sub> ).
13	0014	27	Tens BCD→ Digit 01	0	BEQ. Check the Z flag. It's clear so continue.
14	0016	4A	01	0	Decrement the tens BCD digit (01 <sub>16</sub> ).
15	0017	97	New Tens→ Digit 00	1	Store the new tens BCD digit.
16	0019	96	00	1	Load the accumulator with the binary result (64 <sub>16</sub> ).
17	001B	8B	64	0	Add 0A <sub>16</sub> to the binary result.
18	001D	97	New Binary→ Result 6E	0	Store away the new binary result.
19	001F	20	6E	0	Branch back to address 0012 <sub>16</sub> .
20	0012	96	6E	0	Load the accumulator with the tens BCD digit (00).
21	0014	27	00	1	BEQ. Check the Z flag. It's set so branch to address 0021 <sub>16</sub> .
22	0021	96	00	1	Load the accumulator with the binary result (6E <sub>16</sub> ).
23	0023	9B	6E	0	Add the units BCD digit (07 <sub>16</sub> ).
24	0025	97	New Binary→ Result 75	0	Store the new binary result (75 <sub>16</sub> ).
25	0027	3E	75	0	Halt.

Figure 9-40

Single-stepping through the BCD-to-binary conversion program.



The most frequent mistake made by programmers when using the branch instructions is the improper computation of the relative address. An improperly coded relative address not only prevents the program from executing properly, but can even wipe out portions of the program. In the next section of this experiment, you will witness the result of an incorrect relative address and the effect it has on the program. In this example, we will use the binary-to-BCD conversion program you studied earlier.

## Procedure (Continued)

28. Load the program listed in Figure 9-41 into the Trainer. This program should convert the binary number  $0111\ 0101_2$  ( $75_{16}$ ) into its BCD equivalent. However, one of the relative addresses is **incorrect**. Part of this exercise is to locate the incorrect relative address and correct it.
29. Now set the program counter to 0000 and single-step through the program. Record the results in the chart of Figure 9-42. Notice that we're monitoring the carry (C) flag because the program uses the BCS (Branch if Carry Set) instruction.
30. Examine addresses  $002B_{16}$ ,  $002C_{16}$ , and  $002D_{16}$ ; record the results below.

002B \_\_\_\_ Hundreds BCD Digit

002C \_\_\_\_ Tens BCD Digit

002D \_\_\_\_ Units BCD Digit

Obviously, there is something wrong with the program. Although the hundreds and tens digits are believable, the units digit of 11 is impossible. Remember, a decimal number can only have a units digit of from 0 to  $9_{10}$ .

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	4F	CLRA	Clear the accumulator.
0001	97	STA	Store 00
0002	2B	2B	in location 002B <sub>16</sub> . This clears the hundreds digit.
0003	97	STA	Store 00.
0004	2C	2C	in location 002C <sub>16</sub> . This clears the tens digit.
0005	97	STA	Store 00
0006	2D	2D	in location 002D <sub>16</sub> . This clears the units digit.
0007	96	LDA	Load direct into the accumulator
0008	2A	2A	the binary number to be converted.
0009	80	SUB	Subtract immediate
000A	64	64	100 <sub>16</sub> .
000B	25	BCS	If a borrow occurred, branch
000C	0A	0A	forward to the instruction in location 0016 <sub>16</sub> .
000D	97	STA	Otherwise, store the result of the subtraction
000E	2A	2A	as the new binary number.
000F	96	LDA	Load direct into the accumulator
0010	2B	2B	the hundreds digit of the BCD result.
0011	4C	INCA	Increment the hundreds digit.
0012	97	STA	Store the hundreds digit
0013	2B	2B	back where it came from.
0014	20	BRA	Branch
0015	F1	F1	back to the instruction at address 0007 <sub>16</sub> .
0016	96	LDA	Load direct into the accumulator
0017	2A	2A	the binary number.
0018	80	SUB	Subtract immediate
0019	0A	0A	10 <sub>16</sub> .
001A	25	BCS	If a borrow occurred, branch
001B	09	09	forward to the instruction in location 0025 <sub>16</sub> .
001C	97	STA	Otherwise, store the result of the subtraction
001D	2A	2A	as the new binary number.
001E	96	LDA	Load direct into the accumulator
001F	2C	2C	the tens digit.
0020	4C	INCA	Increment the tens digit.
0021	97	STA	Store the tens digit.
0022	2C	2C	back where it came from.
0023	20	BRA	Branch
0024	F1	F1	back to the instruction at address 0016 <sub>16</sub> .
0025	96	LDA	Load direct into the accumulator
0026	2A	2A	the binary number.
0027	97	STA	Store it in
0028	2D	2D	the units digit.
0029	3E	HLT	Halt.
002A	75	75	Place binary number to be converted at this address.
002B	—	—	Hundreds digit
002C	—	—	Tens digit
002D	—	—	Units digit

 Reserved for  
 BCD result.

Figure 9-41

A program with an incorrect relative address.

STEP	PROGRAM COUNTER	OPCODE	ACCA	C FLAG	COMMENTS
1					
2					
3					
4					
5					
6					
7					
8					
9					
10					
11					
12					
13					
14					
15					
16					
17					
18					
19					
20					

Figure 9-42

Single-Stepping through the binary-to-BCD conversion program.

31. Use the program listing and the chart that you've compiled and locate the error in the program. Then record the address of the instruction below.

HINT: The problem is with the relative address for one of the branch instructions. When one of these addresses is incorrect, the program branches to the wrong address, possibly skipping portions of the program. Therefore, first determine the portions of the program that produced the wrong result and work back until you find the problem.

Address — — — — Incorrect Relative Address — —

32. Now calculate the correct relative address (operand) and record it below.

Correct Relative Address — —.

## Discussion

This exercise should have demonstrated the versatility of your Trainer to assist you in “debugging” programs. When you examined addresses  $002B_{16}$ ,  $002C_{16}$ , and  $002D_{16}$ , you found these results.

002B    0   1        Hundreds BCD Digit

002C    0   0        Tens BCD Digit

002D    1   1        Units BCD Digit

Obviously, the units BCD digit is incorrect. Since the units digit is wrong, we begin to debug at this portion of the program. This happens to be the least complex section of the program because the binary number is simply loaded into the accumulator and stored in address  $002D_{16}$ . Comparing the chart that you compiled against the program listing, we find that this portion of the program seems to be executing correctly.

Therefore, we move back to the tens BCD digit portion of the program. Checking the program listing, we find that the tens BCD portion of the program begins at address  $0016_{16}$ . But as the chart in Figure 9-43 shows, when the program is single-stepped the tens BCD digit loop actually starts at address  $0017_{16}$ . This is the wrong address. We find the problem when we move back to step 14 of the chart. This is the BCS (Branch if Carry Set) instruction at address  $000B_{16}$ . However, instead of branching to address  $0016_{16}$  as the comments column suggests, the program goes to address  $0017_{16}$ . Therefore, the relative address at address  $000C_{16}$  must be incorrect. When we check this relative address, we find that it should be  $09_{16}$ , instead of  $0A_{16}$ .

But, how did this incorrect operand affect the program? Following the chart in Figure 9-43, we find that the hundreds BCD portion of the program worked correctly. On the second loop through this portion of the program, the subtraction resulted in a borrow and the C flag was set. Hence, the BCS instruction produced the desired branch.

STEP	PROGRAM COUNTER	OPCODE	ACCA	C FLAG	COMMENTS
1	0001	97	00	0	Store 00 in Hundreds Digit.
2	0003	97	00	0	Store 00 in tens Digit.
3	0005	97	00	0	Store 00 in units Digit.
4	0007	96	00	0	Load the accumulator with the Binary number (75 <sub>16</sub> ).
5	0009	80	75	0	Subtract 64 <sub>16</sub> from accumulator
6	000B	25	11	0	BCS. Check C flag for borrow. It's clear so continue.
7	000D	97	11	0	Store away the new binary number.
8	000F	96	11	0	Load the accumulator with the Hundreds Digit (00).
9	0011	4C	00	0	Increment the Hundreds Digit.
10	0012	97	01	0	Store the Hundreds Digit.
11	0014	20	01	0	Branch back to address 0007 <sub>16</sub> .
12	0007	96	01	0	Load the accumulator with the Binary Number (11 <sub>16</sub> ).
13	0009	80	11	0	Subtract 64 <sub>16</sub> from accumulator. BCS. Check C Flag for borrow.
14	000B	25	AD	1	It's set so branch to address 0016 <sub>16</sub> .
15	<div>Tens BCD</div> 0017	<div>Wrong Address 2A</div> 25	AD	1	What's this?
16	0019	0A	AD	1	
17	001A	25	AD	1	BCS. Check C Flag. It's still set so branch to address 0025 <sub>16</sub> .
18	<div>Units BCD</div> 0025	96	AD	1	Load the accumulator with the Binary number.
19	0027	97	11	1	Store it in the units Digit.
20	0029	3E	11	1	Halt.

Figure 9-43

Locating the incorrect relative address.

But, instead of branching to address  $0016_{16}$ , where we would have found a load accumulator instruction ( $96_{16}$ ) with an operand of  $2A_{16}$ , the program branches to address  $0017_{16}$ . The Trainer now interprets the operand ( $2A_{16}$ ) as an instruction or op code. The op code  $2A$ , as you may recall, represents a valid instruction which is "Branch if Plus." The MPU checks the N flag and finds it set, because at this time, the negative number  $AD_{16}$  is in the accumulator. Therefore, the condition is not satisfied, and the Trainer continues on to the next instruction.

Single-stepping again (now we are at step 16) the next op code is  $0A$ . Actually, this should be the operand for the subtract instruction at address  $0018_{16}$ . But since we are off by one, it appears to be the op code. The Trainer checks the op code  $0A$  and finds that it's an inherent instruction to "clear the overflow flag." It executes this instruction.

Step 17 finds the program at address  $001A_{16}$ . Here, we encounter another BCS conditional branch instruction. The C flag is still set so we branch to address  $0025_{16}$ . The program works properly from this point on.

Therefore, this one incorrect relative address caused the program to skip the tens BCD portion of the program. The tens unit was never subtracted, so it carried over into the units BCD digit. This produced the wrong units digit of  $11_{10}$ .

## Procedure (Continued)

33. Now change the operand at address  $000C_{16}$  from  $0A_{16}$  to  $09_{16}$ .
34. Also change the number at address  $002A_{16}$  to  $75_{16}$ . This is the number that the program will convert to its BCD equivalent.
35. Reset the program counter to  $0000$  and single-step through the program comparing the program listing with the results that you obtain.

36. Examine the addresses listed below and record the information stored there.

002B    —    Hundreds BCD Digit

002C    —    Tens BCD Digit

002D    —    Units BCD Digit

Is this the correct BCD representation for the number  $75_{16}$ ?

\_\_\_\_\_.

## Discussion

When the program is corrected by inserting the relative address ( $09_{16}$ ) at address  $000C_{16}$ , we find that it works perfectly. After single-stepping through the program, we examine the BCD digits stored at addresses  $002B_{16}$ ,  $002C_{16}$ , and  $002D_{16}$ . The hundreds digit is  $01_{10}$ , the tens digit is  $01_{10}$ , and the units digit is  $07_{10}$ . Therefore, the BCD equivalent of the binary number  $0111\ 0101_2$  ( $75_{16}$ ) is  $117_{10}$ .

## Experiment 6

### ADDITIONAL INSTRUCTIONS

#### OBJECTIVES:

*To verify the operation of the ADC instruction when used in a multiple-precision addition program.*

*To investigate the hazard of using the ADC instruction when a carry is not desired.*

*To demonstrate your ability to write a multiple-precision subtraction program using the SBC instruction.*

*To demonstrate your ability to write a routine that will multiply any 4-bit binary number times  $16_{10}$  using the ASLA instruction.*

*To verify the operation of a BCD packing program that uses the ASLA instruction.*

*To verify the operation of the DAA instruction when used in a BCD multiple-precision addition program.*



## Introduction

One of the measures of a microprocessor's power is the size of the instruction set. In other words, more instructions generally mean more potential power. You saw the economy that resulted with the addition of branch instructions in the previous experiment. In this experiment, we will examine four additional instructions; the ADC or add with carry, the SBC or subtract with carry, the ASLA or arithmetic shift accumulator left, and the DAA or decimal adjust accumulator.

The discussion in Unit 4 explained the purpose of each instruction. In this experiment, we will restrict our activity to verifying that each instruction works as explained.

In the previous experiment, you examined the condition code registers and how the MPU monitors these flag registers to initiate conditional branches. Yet, these condition code registers are also monitored for other instructions. For example, the ADC (add with carry) and SBC (subtract with carry) instructions key on the C or carry flag. If an ADC instruction is executed and the carry flag is set, one is added to the least significant bit in the accumulator. Likewise, if the C flag is set when an SBC instruction is executed, one is subtracted from the least-significant bit of the accumulator. Remember, the C flag represents a "borrow" to the subtract instruction.

In the first portion of this experiment, we will verify the operation of the ADC instruction with a program for multiple precision arithmetic. Then we will examine one of the hazards of using this instruction.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	01	NOP	No operation
0001	96	LDA	Load the accumulator direct with the
0002	0E	OE	least significant byte of the addend.
0003	9B	ADD	Add direct the
0004	10	10	least significant byte of the augend.
0005	97	STA	Store the result in the
0006	12	12	least significant byte of the sum.
0007	96	LDA	Load the accumulator direct with the
0008	0F	OF	most significant byte of the addend.
0009	99	ADC	Add with carry direct the
000A	11	11	most significant byte of the augend.
000B	97	STA	Store the result in the
000C	13	13	most significant byte of the sum.
000D	3E	HLT	Halt
000E	EA	EA	Least significant byte
000F	CO	CO	Most significant byte
			} addend
0010	93	93	Least significant byte
0011	1B	1B	Most significant byte
			} augend
0012	—	—	Least significant byte
0013	—	—	Most significant byte
			} sum

Figure 9-44

Program for multiple-precision addition.

## Procedure

1. Turn on the Trainer and press the RESET key.
2. Load the program listed in Figure 9-44 into the Trainer. This program performs multiple-precision addition of two  $16_{10}$  bit numbers. The augend  $1B93_{16}$  will be added to the addend  $COEA_{16}$  by this program. Of course, the program can add any numbers that are  $16_{10}$  bits or less.
3. Change the program counter to 0000 and single-step through the program, recording the information in the chart of Figure 9-45. Notice that we are monitoring the carry (C) flag.
4. Examine memory location  $0012_{16}$  and  $0013_{16}$  and record the sum below.

SUM \_ \_ \_ \_

STEP	PROGRAM COUNTER	OPCODE	ACCA	C FLAG	COMMENTS
1					
2					
3					
4					
5					
6					
7					

Figure 9-45

5. Add the binary numbers below. These numbers are the binary equivalent of the two hex numbers added by the program just executed.

		MSB		LSB
$COEA_{16}$	=	1100	0000	1110 1010
$1B93_{16}$	=	0001	1011	1001 0011
SUM	=			

Now, convert the binary sum to its hexadecimal equivalent and record below.

SUM \_ \_ \_ \_

Does this match the sum obtained in step 4? \_\_\_\_\_

6. Now load the program of Figure 9-46 into the Trainer. This program simply adds two binary numbers and produces a carry. Hence, it will set the C flag. You will see its purpose in a moment.

Execute the program by pressing the DO key and then entering address 0000.

7. Examine the carry (C) condition code register. The C flag is \_\_\_\_\_

set/reset

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	86	LDA	Load the accumulator immediate with $EA_{16}$ .
0001	EA	EA	
0002	8B	ADD	Add immediate
0003	93	93	93
0004	3E	HLT	Halt

Figure 9-46

Program adds two numbers and produces carry.

8. Enter the program listed in Figure 9-47 into the Trainer. Notice that this is the same multiple-precision addition program previously executed, with the exception that the ADD Instruction has been replaced by the ADC instruction, as shown by the shaded section.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	01	NOP	No operation
0001	96	LDA	Load the accumulator direct with the
0002	0E	0E	least significant byte of the addend
0003	99	ADC	Add with carry direct the
0004	10	10	least significant byte of the augend.
0005	97	STA	Store the result in the
0006	12	12	least significant byte of the sum.
0007	96	LDA	Load the accumulator direct with the
0008	0F	0F	most significant byte of the addend.
0009	99	ADC	Add with carry direct the
000A	11	11	most significant byte of the augend.
000B	97	STA	Store the result in the
000C	13	13	most significant byte of the sum.
000D	3E	HLT	Halt
000E	EA	EA	Least significant byte
000F	CO	CO	Most significant byte
			} addend
0010	93	93	Least significant byte
0011	1B	1B	Most significant byte
			} augend
0012	—	—	Least significant byte
0013	—	—	Most significant byte
			} sum

Figure 9-47

 Multiple-precision addition program with instruction at address 0003<sub>16</sub> changed.

- 9. Set the program counter to 0000 and single-step through the program, recording the information in the chart of Figure 9-48.
- 10. Examine memory locations 0012<sub>16</sub> and 0013<sub>16</sub>. Record the sum below.

SUM \_ \_ \_ \_

Compare this sum to the previous sum recorded in step 4. Are they the same? \_\_\_\_\_.  
yes/no

Why are the sums different? \_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_

From this demonstration, what conclusion can you draw concerning the use of the ADC instruction? \_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_

STEP	PROGRAM COUNTER	OPCODE	ACCA	C FLAG	COMMENTS
1					
2					
3					
4					
5					
6					
7					

Figure 9-48

## Discussion

In steps 1 through 3 of this experiment, you loaded a multiple-precision addition program similar to the one you studied in Unit 4. Single-stepping through the program, you witnessed the operation of the ADC instruction. The chart you compiled should be similar to the chart in Figure 9-49. When you checked memory locations 0012<sub>16</sub> and 0013<sub>16</sub>, you found the LSB and MSB respectively of the 16<sub>10</sub>-bit sum. The sum should have been DC7D<sub>16</sub>.

In step 5 you added the binary equivalents of the hex numbers, COEA<sub>16</sub> and 1B93<sub>16</sub>. The sum was the binary equivalent of the sum produced by the program, as shown below.

		MSB	LSB
		1	
COEA <sub>16</sub>	=	1100 0000	1110 1010
1B93 <sub>16</sub>	=	0001 1011	1001 0011
SUM	=	1101 1100	0111 1101

As you noticed, a carry is generated by the addition of the least significant bytes of the two numbers. When you were single-stepping through the program, you observed this carry because the C flag was set. The addition of the most significant bytes did not produce a carry. Therefore, the carry flag was cleared.

STEP	PROGRAM COUNTER	OPCODE	ACCA	C FLAG	COMMENTS
1	0001	96	Random	Random	Load the accumulator with the LSB of Addend (EA <sub>16</sub> ).
2	0003	9B	EA	Random	Add the LSB of the Augend (93 <sub>16</sub> ).
3	0005	97	7D	1	Store result in LSB of sum.
4	0007	96	7D	1	Load the accumulator with the MSB of the Addend (CO <sub>16</sub> ).
5	0009	99	CO	1	Add with carry the MSB of the Augend (1B <sub>16</sub> ).
6	000B	97	DC	0	Store result in MSB of Sum.
7	000D	3E	DC	0	Halt.

Figure 9-49

When you converted the binary number to hexadecimal, you found that the sum was the same as that produced by the program.

1101 1100      0111 1101

D      C              7      D

In step 6, you loaded a simple program that added the numbers  $EA_{16}$  and  $93_{16}$ . Of course, the addition generated a carry, as you witnessed when you checked the C flag and found it set.

In step 8, you loaded another multiple-precision addition program into the Trainer. The only difference between this program and the previous multiple-precision addition program was that the first add instruction was the ADC (add with carry), rather than the ADD. Then you single-stepped through the program and completed the chart of Figure 9-48. Your chart should be similar to the one shown in Figure 9-50.

When you examined the sum at addresses  $0012_{16}$  and  $0013_{16}$ , you found  $DC7E_{16}$ . The correct sum, as you verified earlier, should have been  $DC7D_{16}$ . If you checked the chart compiled while single-stepping through the program, the reason for this incorrect answer should have been evident. The carry flag was set even before the program was executed. Therefore, when the Trainer executed the first ADC instruction, it automatically added the carry ( $1_2$ ) to the sum of the least significant bytes. Hence, the result  $7E$  was one greater than the correct sum of  $7D$ .

STEP	PROGRAM COUNTER	OPCODE	ACCA	C FLAG	COMMENTS
1	0001	96	Random	1	Load the accumulator with the LSB of Addend ( $EA_{16}$ ).
2	0003	99	EA	1	Add with carry the LSB of the Augend $93_{16}$ .
3	0005	97	7E	1	Store result in LSB of sum. Load the accumulator with the MSB
4	0007	96	7E	1	of Addend ( $CO_{16}$ ).
5	0009	99	CO	1	Add with carry the MSB of the Augend ( $1B_{16}$ ).
6	000B	97	DC	0	Store result in MSB of sum.
7	000D	3E	DC	0	Halt.

Figure 9-50

Single-stepping through the multiple-precision addition program where both add instructions are ADC.



From this demonstration you should have reached the conclusion that the ADC instruction should not be used unless you are positive of the condition of the C flag. You must remember that the C flag is only reset by an arithmetic operation that doesn't produce a **carry** or a **borrow**. For example, in the program that worked properly, we used the simple ADD instruction for the first addition. Naturally, this instruction ignores the condition of the C flag, so it doesn't matter if it's set or reset. This is a simple way of playing it safe. The second addition used the ADC instruction because we wanted any carry from the least significant byte to be reflected in the most significant byte.

The SBC (subtract with carry) instruction is similar to the ADC instruction because it also monitors the C flag to indicate a borrow. In the next section of this experiment, you will write a program that uses the SBC instruction for multiple-precision subtraction of  $16_{10}$ -bit numbers.

## Procedure (Continued)

11. Write a program that will perform multiple-precision subtraction of two  $16_{10}$ -bit (2-byte) numbers. The following guidelines define the problem.
  - a. The program must subtract a  $16_{10}$ -bit subtrahend from a  $16_{10}$ -bit minuend and store the difference in memory.
  - b. Use the direct addressing mode.
  - c. Select the op codes from the instruction listing in Figure 9-51.
12. Now load the program. Enter  $9721_{16}$  in the locations reserved for the minuend and  $7581_{16}$  in the locations reserved for the subtrahend.
13. Single-step through the program and observe its operation. Examine the locations where the difference is stored and record the 2-byte difference below.

DIFFERENCE \_\_\_\_\_

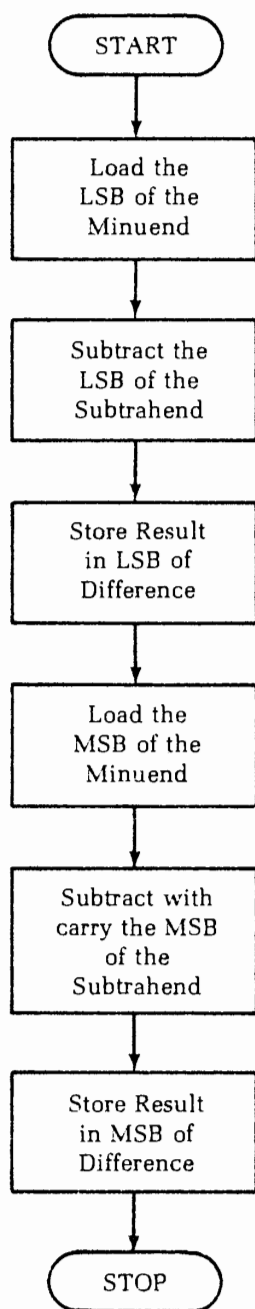


Figure 9-52  
Flow chart for  
multiple-precision subtraction.

INSTRUCTION	MNEMONIC	ADDRESSING MODE			
		IMMEDIATE	DIRECT	RELATIVE	INHERENT
Load Accumulator	LDA	86	96		
Clear Accumulator	CLRA				4F
Decrement Accumulator	DECA				4A
Increment Accumulator	INCA				4C
Store Accumulator	STA		97		
Add	ADD	8B	9B		
Subtract	SUB	80	90		
Add with Carry	ADC	89	99		
Subtract with Carry	SBC	82	92		
Arithmetic Shift Accumulator Left	ASLA				48
Decimal Adjust Accumulator	DAA				19
Halt	HLT				3E

Figure 9-51  
Instructions.

## Discussion

If you made a flow chart of the problem, your flow chart probably looks like the one shown in Figure 9-52. Your program should be similar to the solution shown in Figure 9-53. After stepping through the program on the Trainer, the difference of the subtraction should have been  $21A0_{16}$ . If you didn't obtain this answer, go back and recheck your program.

You may have used the SBC instruction for the first subtraction. If you did, this might explain the problem, because if the C flag is set when this instruction is executed a 1 will be borrowed from the difference. Therefore, your answer would have been 1 less than the correct answer, or  $219F_{16}$ . If the carry flag was cleared before you executed the program, the result would still be correct.

In the next section of this experiment, we will examine the ASLA (arithmetic shift accumulator left) instruction. You will also write a simple program that uses this instruction to multiply any  $4_{10}$ -bit number by  $16_{10}$ . This simple routine will prove it's usefulness later.

Recall from the discussion in Unit 4 that each ASLA operation multiplies the contents of the accumulator by two.

## Procedure (Continued)

14. Use the instructions listed in Figure 9-51 and write a program that uses the ASLA instruction to multiply any  $4_{10}$ -bit number by  $16_{10}$ .

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	96	LDA	Load accumulator direct with
0001	0D	0D	least significant byte of minuend
0002	90	SUB	Subtract direct
0003	0F	0F	least significant byte of subtrahend
0004	97	STA	Store result in
0005	11	11	least significant byte of difference
0006	96	LDA	Load accumulator direct with
0007	0E	0E	most significant byte of minuend
0008	92	SBC	Subtract with carry
0009	10	10	most significant byte of the subtrahend
000A	97	STA	Store result in
000B	12	12	most significant byte of difference
000C	3E	HLT	Halt
000D	21	21	Least significant byte
000E	97	97	Most significant byte
000F	81	81	Least significant byte
0010	75	75	Most significant byte
0011	—	—	Least significant byte
0012	—	—	Most significant byte

Figure 9-53

Program for multiple-precision subtraction.

15. Enter your program into the Trainer and then have your program multiply  $0F_{16}$  ( $15_{10}$ ) by  $16_{10}$ . Record the product below.

$$0F_{16} \times 16_{10} = \text{————}_{16}$$

16. Convert the product obtained to its decimal equivalent.

Decimal equivalent ————<sub>10</sub>.

Now check your result by multiplying  $15_{10}$  times  $16_{10}$ .

$$15_{10} \times 16_{10} = \text{————}_{10}$$

17. In this program, the multiplier is determined by the number of ASLA instructions. How many ASLA instructions are required to produce a multiplier of  $4_{10}$ ? ————.

## Discussion

The program for this simple routine is shown in Figure 9-54. Notice that it uses  $4_{10}$  ASLA instructions to produce the required multiplier of  $16_{10}$ . If your program worked properly, the final product should have been  $F0_{16}$ . Converting this number to its decimal equivalent, we find that  $F0_{16}$  equals  $240_{10}$ . When we multiplied  $15_{10}$  times  $16_{10}$ , we also found the product was  $240_{10}$ . Therefore, the program works.

Only two ASLA instructions are necessary to produce a multiplier of  $4_{10}$ ; three ASLA instructions will result in a multiplier of  $8_{10}$ .

Another use for the ASLA instruction is to pack two BCD digits into a single byte. This "packing" can result in a significant savings of memory if many BCD numbers are used. Let's verify the operation of the BCD packing program that was presented in Unit 4.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	96	LDA	Load the accumulator with the 4-bit multiplicand
0001	09	09	
0002	48	ASLA	} Shift the accumulator four places to the left multiplying the multiplicand by $16_{10}$ .
0003	48	ASLA	
0004	48	ASLA	
0005	48	ASLA	
0006	97	STA	Store the product at this location
0007	0A	0A	
0008	3E	HLT	Halt
0009	0F	0F	4-bit multiplicand
000A	—	—	Product

Figure 9-54

Program that uses the ASLA instruction to multiply a 4-bit number times  $16_{10}$ .

## Procedure (Continued)

18. Enter the BCD packing program listed in Figure 9-55 into the Trainer. The unpacked BCD numbers are 09<sub>10</sub> and 03<sub>10</sub>.
19. Set the program counter to 0000 and single-step through the program, recording the information below. Where it is indicated, convert the hexadecimal contents of the accumulator to the binary equivalent.

Program Count	Op code	ACCA	Binary Equivalent
0001	96	Random	Random
0003	48	_____	_____
0004	48	_____	_____
0005	48	_____	_____
0006	48	_____	_____
0007	9B	_____	_____
0009	97	_____	_____
000B	3E	HALT	

HEX ADDRESS	OPCODES/ CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	01	NOP	Do nothing
0001	96	LDA	Load into the accumulator direct
0002	0D	0D	the unpacked most significant BCD digit.
0003	48	ASLA	} Shift it four places to the left.
0004	48	ASLA	
0005	48	ASLA	
0006	48	ASLA	
0007	9B	ADD	Add the
0008	0E	0E	unpacked least significant BCD digit.
0009	97	STA	Store the result
000A	0C	0C	in the packed BCD number
000B	3E	HLT	Halt
000C	00	00	Packed BCD number
000D	09	09	Unpacked most significant BCD digit.
000E	03	03	Unpacked least significant BCD digit.

Figure 9-55

Program to pack two BCD digits into a single byte.

20. Examine the packed BCD number at address 000C<sub>16</sub> and record it below.

Packed BCD Number \_\_\_\_\_

## Discussion

As you can see, the BCD packing program is very simple. Nevertheless, simple routines such as this can be combined in many programs, easing the task of programming. Most programmers either commit these general purpose routines to memory or file them away for future reference.

The results you obtained by stepping through the program should be similar to those shown below.

PROGRAM COUNT	OP CODE	ACCA	BINARY EQUIVALENT
0001	96	Random	Random
0003	48	09	0000 1001
0004	48	12	0001 0010 After 1st shift
0005	48	24	0010 0100 After 2nd shift
0006	48	48	0100 1000 After 3rd shift
0007	9B	90	1001 0000 After 4th shift
0009	97	93	1001 0011
000B	3E		

As the listing shows, the most significant BCD digit (09<sub>10</sub>) is loaded into the accumulator. Four ASLA shifts take place, moving this digit progressively to the left. Following these four shifts, the most significant BCD digit is properly positioned. Now the program simply adds the least significant BCD (03<sub>10</sub>) to the contents of the accumulator and then stores the sum. Checking the address of the packed BCD number, we find 93<sub>10</sub>.

When BCD numbers are added, we encounter yet another problem. Often, the sum is the correct BCD number. But, just as frequently, it isn't. In Unit 4, the reason for this inconsistency was discussed. However, your Trainer has an instruction, called the "Decimal Adjust Accumulator" (DAA), that can correct the sum of BCD numbers, producing the desired result.

In the next portion of this experiment, we will demonstrate the need for the DAA instruction by first adding two BCD numbers without using the DAA instruction. Then we will check the sum. Next, we will correct the program by inserting DAA instructions and again examine the BCD sum.

### Procedure (Continued)

21. Load the program listed in Figure 9-56 into your Trainer. This program adds the BCD numbers  $3792_{10}$  and  $5482_{10}$ , storing the sum in address  $0011_{16}$  and  $0012_{16}$ .
22. RESET the Trainer and execute the program by first pressing the DO key and entering address 0000.
23. Again, press the RESET key and then examine the sum stored at address  $0011_{16}$  and  $0012_{16}$ . The most significant byte of the sum is at address  $0011_{16}$  and the least significant byte is at address  $0012_{16}$ . Record the sum below.

SUM \_\_\_\_\_

**Is this the correct BCD sum for the addition of the numbers  $3792_{10}$  and  $5482_{10}$ ? \_\_\_\_\_.**

yes/no

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	96	LDA	Load the accumulator direct with
0001	0E	0E	the least significant byte of addend.
0002	9B	ADD	Add direct
0003	10	10	the least significant byte of augend
0004	97	STA	Store the result in
0005	12	12	the least significant byte of BCD sum.
0006	96	LDA	Load the accumulator direct with
0007	0D	0D	the most significant byte of addend
0008	99	ADC	Add with carry
0009	0F	0F	the most significant byte of augend
000A	97	STA	Store the result in
000B	11	11	the most significant byte of BCD sum.
000C	3E	HLT	Halt
000D	37	37	Most significant byte
000E	92	92	Least significant byte
000F	54	54	Most significant byte
0010	82	82	Least significant byte
0011	—		Most significant byte
0012	—		Least significant byte

**Figure 9-56**

Incorrect program for multiple-precision addition of BCD numbers.

24. Now load the corrected multiple-precision BCD addition program listed in Figure 9-57 into your Trainer. Notice that the only changes between this program and the previous program are the additions of the NOP instruction and the two DAA instructions following the addition operations.
25. Change the program counter to 0000 and single-step through the program, recording the information below.

STEP 1

PROGRAM COUNT

OP CODE

STEP 2

PROGRAM COUNT

OP CODE

ACCA

STEP 3

PROGRAM COUNT

OP CODE

ACCA

C FLAG

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	01	NOP	Do nothing
0001	96	LDA	Load the accumulator direct with the
0002	11	11	least significant byte of addend.
0003	9B	ADD	Add direct
0004	13	13	the least significant byte of augend.
0005	19	DAA	Decimal adjust the sum to BCD.
0006	97	STA	Store the result in the
0007	15	15	least significant byte of BCD sum
0008	96	LDA	Load the accumulator direct with the
0009	10	10	most significant byte of addend.
000A	99	ADC	Add with carry the
000B	12	12	most significant byte of augend.
000C	19	DAA	Decimal adjust the sum to BCD.
000D	97	STA	Store the result in the
000E	14	14	most significant byte of BCD sum.
000F	3E	HLT	Halt.
0010	37	37	Most significant byte
0011	92	92	Least significant byte
0012	54	54	Most significant byte
0013	82	82	Least significant byte
0014	—	—	Most significant byte
0015	—	—	Least significant byte

Figure 9-57

Program for adding multiple-precision BCD numbers.



The sum of the addition of the least significant bytes is now in the accumulator. Is this the correct BCD sum for the numbers  $92_{10}$  and  $82_{10}$ ? \_\_\_\_\_  
yes/no

When the DAA instruction (op code 19) is executed, will this number be corrected? \_\_\_\_\_.

yes/no

STEP 4      PROGRAM COUNT      OP CODE      ACCA      C FLAG

As you can see, the DAA instruction did correct the left-most digit by adding  $60_{16}$  to the sum. Since the result  $14_{10}$  appears to be a legitimate BCD number, how did the MPU know it was not the valid BCD sum? \_\_\_\_\_

STEP 5	<u>PROGRAM COUNT</u>	<u>OP CODE</u>	<u>ACCA</u>	<u>C FLAG</u>
--------	----------------------	----------------	-------------	---------------

STEP 6	<u>PROGRAM COUNT</u>	<u>OP CODE</u>	<u>ACCA</u>	<u>C FLAG</u>

STEP 7	<u>PROGRAM COUNT</u>	<u>OP CODE</u>	<u>ACCA</u>	<u>C FLAG</u>
--------	----------------------	----------------	-------------	---------------

It's obvious that this number ( $8C_{16}$ ) is not the BCD sum of  $37_{10}$  and  $54_{10}$ . What number will the MPU add to  $8C_{16}$  to produce the desired BCD sum? \_\_\_\_\_.

STEP 8	<u>PROGRAM COUNT</u>	<u>OP CODE</u>	<u>ACCA</u>	<u>C FLAG</u>
--------	----------------------	----------------	-------------	---------------

STEP 9	<u>PROGRAM COUNT</u>	<u>OP CODE</u>	<u>ACCA</u>

26. Now examine the BCD sum at addresses 0014<sub>16</sub> and 0015<sub>16</sub> and record below.

SUM \_\_\_\_\_<sub>10</sub>

## Discussion

When you executed the first program to add BCD numbers, it was obvious that the sum 8C14 was not the correct BCD number. The answer should have been 9274<sub>10</sub>. Naturally, the MPU considered these BCD numbers as hexadecimal numbers, hence, the hexadecimal sum.

However, when the program was modified by the addition of DAA (decimal adjust accumulator) instructions after each addition operation, the result was the correct BCD number. As you stepped through the program you saw the DAA instruction in operation.

At step 3, the BCD numbers 92<sub>10</sub> and 82<sub>10</sub> had been added and the accumulator was supposedly storing the sum 14<sub>10</sub>. A carry was generated by the setting of the C flag. However, the sum was not correct. Instead of 14<sub>10</sub>, the sum should have been 174<sub>10</sub>. To the MPU, the addition looked something like this.

	1001	0010 <sub>2</sub>	= 92 <sub>16</sub>
C FLAG	1000	0010 <sub>2</sub>	= 82 <sub>16</sub>
<hr/>			
1 Carry	0001	0100 <sub>2</sub>	114 <sub>16</sub>

If we ignore the carry, the sum 14<sub>16</sub> appears to be a legitimate BCD number. Nevertheless, the sum would be incorrect. Taking the carry flag into consideration, remember it's just an extension of the accumulator, we find the sum is 114<sub>16</sub>. In hex, this is the correct sum of the two numbers.

In step 4, the DAA instruction had been executed and, as you witnessed, the number  $14_{16}$  had been adjusted to the correct BCD sum of  $74_{10}$ . The carry flag was set, indicating that the sum of the two left-most 4-bit binary numbers was larger than  $1001_2$  ( $9_{16}$ ). Actually, it was  $1\ 0001_2$ . When the DAA instruction was executed, the MPU followed the conversion rules and adjusted the sum by adding  $60_{16}$  as shown below.

Carry				Carry		
1	0001	0100 <sub>2</sub>	=	1	14 <sub>16</sub>	
	0110	0000 <sub>2</sub>	=		60 <sub>16</sub>	
<hr/>				<hr/>		
1	0111	0100 <sub>2</sub>	=	1	74 <sub>16</sub>	

The result is  $74_{16}$  with a carry of  $1_{16}$ . This is the correct BCD sum for the two BCD numbers. If we include the carry, the result is  $174_{10}$  which is indeed the decimal sum of  $92_{10}$  and  $82_{10}$ . However, this exceeds the capacity of our storage locations, since they're only 8-bits long, so the carry is carried forward to the addition of the most significant bytes of the numbers in the next step.

As you continued single-stepping through the program, the most significant bytes were loaded and added with the ADC instruction. At step 7, the sum of this addition was in the accumulator. It was obvious that the sum  $8C_{16}$  wasn't a BCD number. To adjust this number to the correct BCD sum,  $06_{16}$  was added by the DAA instruction. The BCD adjusted sum  $92_{10}$  was the result.

In the final step of the experiment, you verified program operation by examining the BCD sum at locations  $0014_{16}$  and  $0015_{16}$ . Here you should have found the sum  $9274_{10}$ .

## Experiment 7

### NEW ADDRESSING MODES

#### OBJECTIVES:

*To demonstrate the extended addressing mode.*

*To demonstrate the indexed addressing mode.*

*To gain experience using the instruction set and registers of the MPU.*

#### NOTES:

1. If the Trainer you are using has a model number ET-3400A, it will not be necessary for you to add the two RAM IC's (listed under Material Required) to your Trainer. After reading the introduction, begin this experiment at Procedure step 6.
2. If the Trainer you are using has a model number ET-3400, check IC locations IC16 and IC17. If these two locations do not contain IC's (2112 Heath number 443-721), begin this experiment at Procedure step 1. If these two locations are equipped with the 2112 IC's begin this experiment at Procedure step 6.

### Material Required

Microprocessor Trainer

2 - 2112-2 IC's (Heath Number 443-721)

### Introduction

In Unit 5, you learned that the MPU has two new addressing modes called extended and indexed addressing. Either of these addressing modes can be used to reach operands anywhere in memory. By contrast, the direct addressing mode can be used only when the operand is in the first 256<sub>10</sub> bytes of memory.

## Procedure

1. Turn your ET-3400 Microprocessor Trainer off and unplug it.
2. Locate the two 2112-2 IC's (Heath number 443-721) that were supplied with this course. Notice that these IC's are packed in conductive foam.

NOTE: These IC's are rugged, reliable components. However, normal static electricity discharged from your body through an IC pin to an object can damage the IC. Install these IC's without interruption as follows:

- A. Remove the IC from its package with both hands.
- B. Hold the IC with one hand and straighten any bent pins with the other hand.
- C. Refer to Figure 9-58. Position the pin 1 end of the IC over the index mark on the circuit board.
- D. Be sure each IC pin is properly started into the socket. Then push the IC down.

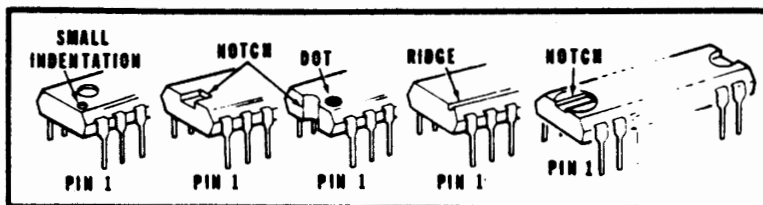


Figure 9-58

3. Install one of the IC's in the empty socket labelled IC16 on the ET-3400 Trainer.
4. Install the other IC in the socket labelled IC17.

NOTE: Until now, you could not use the extended addressing mode because the ET-3400 Trainer had only 256<sub>10</sub> bytes of RAM memory. The installation of the two RAM IC's in the above steps has added an additional 256<sub>10</sub> bytes of RAM memory necessary for the extended addressing mode.



5. Plug in your Trainer and turn it on.
6. Using the AUTO mode, load the numbers 00 through 0F into memory locations 0100 through 010F, respectively.
7. Using the EXAM and FWD keys, verify that the above numbers were stored in those addresses.

## Discussion

The ET-3400A Trainer required no hardware modifications to acquire  $512_{10}$  bytes of RAM in addresses  $0000_{16}$  through  $01FF_{16}$ . The two 2114 RAM IC's at IC14 and IC15 already have this capacity. However, the ET-3400 Trainer uses 2112 RAM IC's. The two IC's at IC14 and IC15 contain only the first  $256_{10}$  bytes of memory from  $0000_{16}$  to  $00FF_{16}$ .

Therefore, to extend the RAM capacity of the ET-3400 Trainer, an additional  $256_{10}$  bytes, it may have been necessary to install two additional 2112 IC's at locations IC16 and IC17. The starting address of this new RAM is  $0100_{16}$  and extends through  $01FF_{16}$  for a total of  $512_{10}$  bytes. When operands are placed at addresses above  $00FF_{16}$ , the extended addressing mode is generally used.

### Procedure (Continued).

8. Figure 9-59 shows a program for adding a list of numbers. Because the numbers are in addresses higher than  $00FF_{16}$ , the extended addressing mode is used. Load this program into the Trainer and verify that you have loaded it properly.
9. Execute the program using the single-step mode. The first instruction sets the contents of accumulator A to \_\_\_\_\_.
10. Examine the program counter and accumulator A after each instruction is executed. Each time an ADDA extended instruction is executed, the program counter is advanced \_\_\_\_\_ bytes.
11. Examine the contents of accumulator A after the final instruction is executed. The number in accumulator A is \_\_\_\_\_.
12. Refer to your instruction set summary card. How many MPU cycles are required to execute this program? \_\_\_\_\_.

### Discussion

The program adds the ten numbers giving the sum  $55_{10}$  or  $37_{16}$ . It requires 51 MPU cycles. Notice that the program itself takes up  $32_{10}$  bytes of memory. The data (the ten numbers) use another  $10_{10}$  bytes.

A repetitive program like this one is an excellent candidate for indexed addressing. Let's see how the same job can be done using indexed addressing.



HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0100	4F	CLRA	Clear accumulator A
0101	BB	ADDA	Add the first number
0102	01	01	which is at this
0103	20	20	address.
0104	BB	ADDA	Add the second number.
0105	01	01	
0106	21	21	
0107	BB	ADDA	Add the third number.
0108	01	01	
0109	22	22	
010A	BB	ADDA	
010B	01	01	
010C	23	23	
010D	BB	ADDA	
010E	01	01	
010F	24	24	
0110	BB	ADDA	
0111	01	01	
0112	25	25	
0113	BB	ADDA	
0114	01	01	Continue until all numbers are added.
0115	26	26	
0116	BB	ADDA	
0117	01	01	
0118	27	27	
0119	BB	ADDA	
011A	01	01	
011B	28	28	
011C	BB	ADDA	
011D	01	01	
011E	29	29	
011F	3E	WAI	Stop.
0120	01	01	First number.
0121	02	02	Second number.
0122	03	03	Third number.
0123	04	04	
0124	05	05	.
0125	06	06	.
0126	07	07	.
0127	08	08	
0128	09	09	
0129	0A	0A	Tenth number.

Figure 9-59

Adding a list of numbers using extended addressing.

## Procedure (Continued)

13. Figure 9-60 shows a program for adding the same list of numbers. However it uses indexed addressing. Load this program into the Trainer and verify that you have loaded it correctly.
14. Execute the program using the single-step mode. After each step, record the contents of the program counter, accumulator A, and the index register in Figure 9-61.
15. Compare the programs of Figures 9-59 and 9-60. Which requires fewer instructions?
16. Refer to the instruction set summary card. How many machine cycles are required to execute the program shown in Figure 9-59 \_\_\_\_\_. Compare this with the number of machine cycles required for the program in Figure 9-60.

## Discussion

This example illustrates that when a repetitive task is to be done, indexed addressing can save many bytes of memory. In many cases, indexed addressing requires more MPU cycles and therefore, a longer time to execute. Generally, time is of little importance compared to saving a substantial number of memory bytes.

Let's look at some other ways that indexed addressing is used.

HEX ADDRESSES	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0130	4F	CLRA	Clear accumulator A
0131	CE	LDX#	Load the index register immediately
0132	01	01	with the address of
0133	20	20	the first number in the list.
0134	AB	→ ADDA, X	Add to accumulator A indexed
0135	00	00	with 00 offset.
0136	08	INX	Increment index register.
0137	8C	CPX#	Compare the index register immediately
0138	01	01	with one greater than the address
0139	2A	2A	of the last number in the list.
013A	26	BNE	If there is no match
013B	F8	F8	branch back to here.
013C	3E	WAI	Otherwise, halt.

Figure 9-60

Adding the list of numbers using indexed addressing.

STEP NUMBER	CONTENTS AFTER EACH STEP		
	PC	ACCA	INDEX
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			
16			
17			
18			
19			
20			
21			
22			
23			
24			
25			
26			
27			
28			
29			
30			
31			
32			
33			
34			
35			
36			
37			
38			
39			
40			
41			
42			
43			

Figure 9-61  
Record values here.

## Procedure (Continued)

17. Write a program that will clear memory locations  $0120_{16}$  through  $01A0_{16}$ . It should use indexed addressing. The program should reside in the lower RAM addresses.
18. When you are sure your program is correct, load it into the ET-3400 Trainer. Verify that you loaded it correctly; then execute it using the DO command.
19. Examine memory locations  $0120_{16}$  through  $01A0_{16}$ . Each should be cleared. Examine locations below  $0120_{16}$  and above  $01A0_{16}$ . These locations should not be cleared.
20. Debug your program if necessary and repeat steps 18 and 19 until the desired results are obtained.

## Discussion

Our solution to the problem is shown in Figure 9-62. Your solution may be similar or quite different. If it achieves the proper result and requires about the same number of bytes, then it is perfectly acceptable.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	CE	LDX#	Load index register immediately with
0001	01	01	the address of the
0002	20	20	first location to be cleared.
0003	6F	CLR, X	Clear the location whose
0004	00	00	address is indicated by the index register.
0005	08	INX	Increment the index register.
0006	8C	CPX#	Compare the number in the index
0007	01	01	register with one greater than
0008	A1	A1	the address of the last location to be cleared.
0009	26	BNE	If there is no match
000A	F8	F8	branch back to here.
000B	3E	WAI	Otherwise, stop.

Figure 9-62

Program for clearing addresses  $0120_{16}$  through  $01A0_{16}$ .

We still have not demonstrated the full power of indexed addressing because we have not yet used the offset capability. Let's look at how the offset capability can be used. Figure 9-63 shows three tables. The first two tables contain signed numbers, the third is initially cleared. The entries in the first two tables are to be added and the resulting sums are to be placed in the third table. That is, the first entry in table 1 is to be added to the first entry in table 2. The resulting sum is to be stored as the first entry of table 3. The second entry in table 1 is to be added to the second entry in table 2, forming the second entry in table 3; etc.

## Procedure (Continued)

21. Enter the data shown in Figure 9-63 into the indicated addresses.
22. Write a program that will solve the problem described above.
23. Enter the program into the Trainer and execute it.
24. Examine addresses 0150<sub>16</sub> through 015F<sub>16</sub> to verify that the program performed properly.
25. If necessary, debug your program and try again.

TABLE 1		TABLE 2		TABLE 3	
ADDRESS	CONTENTS	ADDRESS	CONTENTS	ADDRESS	CONTENTS
0100	06	0110	FA	0150	00
0101	0F	0111	01	0151	00
0102	06	0112	1A	0152	00
0103	20	0113	10	0153	00
0104	2F	0114	11	0154	00
0105	00	0115	50	0155	00
0106	2F	0116	31	0156	00
0107	61	0117	0F	0157	00
0108	3E	0118	42	0158	00
0109	4F	0119	41	0159	00
010A	91	011A	0F	015A	00
010B	9F	011B	11	015B	00
010C	C0	011C	00	015C	00
010D	84	011D	4C	015D	00
010E	70	011E	70	015E	00
010F	E1	011F	0F	015F	00

Figure 9-63

Three tables.

## Discussion

The solution to the problem is shown in Figure 9-64.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	CE	LDX #	Load index register with address
0001	01	01	of first entry
0002	00	00	in Table 1.
0003	A6	LDAA, X	Load entry from Table 1 into
0004	00	00	accumulator A.
0005	AB	ADDA, X	Add the corresponding entry from
0006	10	10	Table 2.
0007	A7	STAA, X	Store the result in the
0008	50	50	corresponding location in Table 3
0009	08	INX	Increment the index register.
000A	8C	CPX #	Compare the number in the index
000B	01	01	register with one greater
000C	10	10	than the address of the last entry in
			Table 1.
000D	26	BNE	If there is no match,
000E	F4	F4	branch to here.
000F	3E	WAI	Otherwise, stop.

Figure 9-64

Program for adding two tables.

## Experiment 8

### ARITHMETIC OPERATIONS

#### OBJECTIVES:

To gain practice using the instruction set and registers of the 6800 MPU.

To demonstrate a fast method of performing multiplication.

To demonstrate multiple-precision arithmetic.

To demonstrate an algorithm for finding the square root of a number.

To gain experience writing programs.

#### Introduction

In Unit 5, you were exposed to the full architecture and instruction set of the 6800 microprocessor. In this experiment, you will use some of the new-found capabilities of the microprocessor to solve some simple problems.

Mathematical operations make excellent programming examples and at the same time illustrate useful procedures. For these reasons, the programs developed in this experiment are concerned with arithmetic operations.

In an earlier unit, you learned that a computer can multiply by repeated addition. However, this is a very slow method of multiplication when large numbers are used.

A much faster method of multiplying involves a shifting-and-adding process. To illustrate the procedure, consider the long hand method of multiplying two 4-bit binary numbers. The procedure looks like this.

1101 <sub>2</sub>	←	Multiplicand	→	13 <sub>10</sub>
1011 <sub>2</sub>	←	Multiplier	→	11 <sub>10</sub>
1101				13
1101				13
0000				143 <sub>10</sub>
1101				
10001111 <sub>2</sub>	←	Product	↘	

The decimal equivalents are shown for comparison purposes. The product is formed by shifting and adding the multiplicand. Put in computer terms, the procedure goes like this:

1. Clear the product.
2. Examine the multiplier. If it is 0, stop. Otherwise, go to 3.
3. Examine the LSB of the multiplier. If it is 1, add the multiplicand to the product then go to 4. If it is a 0, go to 4 without adding.
4. Shift the multiplicand to the left.
5. Shift the multiplier to the right so that the next bit becomes the LSB.
6. Go to 2.

## Procedure

1. Write a program of any length that will perform multiplication in the manner indicated. Here are some guidelines:
  - A. You may use any of the instructions discussed up to this point.
  - B. To keep the program simple, only unsigned 4-bit binary numbers are to be used for the multiplier and the multiplicand.
  - C. The final product should be in Accumulator A when the multiplication is finished.
  - D. The multiplier may be destroyed during the multiplication process.
  - E. Assume that the multiplier and multiplicand are initially in memory. That is, you should load them into memory along with the program.



2. Try to write the program before you read further. If after 30 minutes, you feel you are not making progress, go on to step 3.
3. If you feel you need help, read over the following hints and then write the program.
  - A. The product should be formed in accumulator A.
  - B. The first step is to clear the product.
  - C. The multiplicand is shifted and added to Accumulator A. Accumulator B is a good place to hold the multiplicand during this process.
  - D. The multiplier can be tested for zero while still in memory by using the TST instruction followed by the BEQ instruction.
  - E. A good way to test the LSB of the multiplier is to shift the multiplier one bit to the right into the carry flag and then test the carry flag with a BCC instruction.
4. Once your program is written, load it into the Trainer and run it. Verify that it works for several different values of multipliers and multiplicands. Debug your program as necessary.

## Discussion

The real test of your program is "Does it work?" If it works, then you have successfully completed this part of the experiment. One solution to the problem is shown in Figure 9-65. Compare your program with this one. If you could not write a successful program, study this program carefully to see how it handles each phase of the operation.

Obviously, this simple program has some serious drawbacks. The chief one is that the product cannot exceed eight bits. Fortunately, the basic procedure can be expanded so that much larger numbers can be handled. The solution is to use two bytes for the product. This will allow products up to  $65,535_{10}$ . In this example, the multiplier will be restricted to eight bits. However, the multiplicand can have up to 16 bits (two bytes) as long as the product does not exceed  $65,535_{10}$ . In an earlier unit, you learned that multiple-precision numbers can be added by a 2-step operation. The least significant (LS) byte of one number is added to the LS byte of the other. Then, the MS byte is added **with carry** to the MS byte of the other. Keep this in mind as you write your program.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0010	4F	CLRA	Set the product to 0.
0011	D6	LDAB	Load accumulator B with the
0012	22	22	multiplicand.
0013	7D	→ TST	Test
0014	00	00	the
0015	23	23	multiplier.
0016	27	BEQ	If it is 0, branch to the
0017	09	09	wait instruction.
0018	74	LSR	Shift the LSB of the
0019	00	00	multiplier to the
001A	23	23	right into the carry flag.
001B	24	BCC	If the carry flag is cleared
001C	01	01	skip the next instruction.
001D	1B	ABA	Add the multiplicand to the
			product.
001E	58	ASLB	Shift the multiplicand to the left.
001F	20	BRA	Branch back and go through again.
0020	F2	F2	
0021	3E	WAI	Wait.
0022	05	Multiplicand	
0023	03	Multiplier	

Figure 9-65

Multiplying by shifting and adding.

The procedure for shifting a multiple-precision value will also come in handy. To shift a 2-byte number to the left, a 2-step procedure like that shown in Figure 9-66 can be used. First, the LS byte is shifted one place to the left into the carry bit by using the ASL instruction. Next the MS byte is rotated to the left. The result is that the 16-bit number has been shifted one bit to the left.

## Procedure (Continued)

5. Write a program that will multiply a double-precision multiplicand times an 8-bit multiplier. Assume that the double-precision product is to be stored in memory locations 0000<sub>16</sub> and 0001<sub>16</sub>. The double-precision multiplicand is initially in addresses 0002<sub>16</sub> and 0003<sub>16</sub>. The 8-bit multiplier is in address 0004<sub>16</sub>.
6. Once again, you should try to write this program. If after 30 minutes or so you are not making progress, read the hints given in step 7.

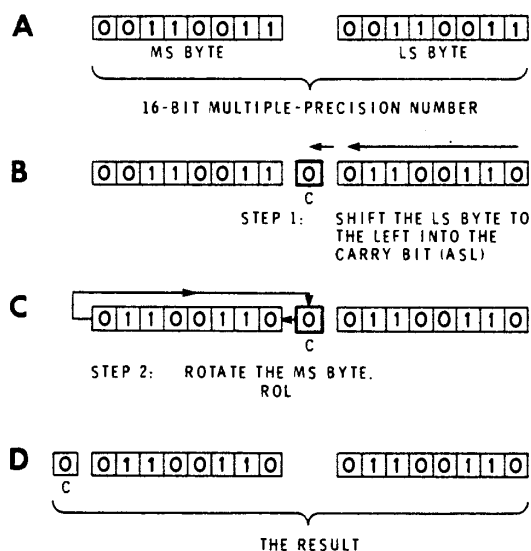


Figure 9-66  
Shifting a multiple-precision number.

7. Read over the following hints (if necessary) and try again.
  - A. Initially clear both bytes of the product.
  - B. Test the multiplier for zero exactly as you did in the previous program.
  - C. Test the LSB of the multiplier as you did in the previous program.
  - D. When adding the multiplicand to the product, use the multiple-precision add technique.
  - E. When shifting the multiplicand to the left, use the technique shown in Figure 9-66.
8. Once your program is written, load it into the Trainer and verify that it works properly. Debug the program as necessary.

## Discussion

There are dozens of ways in which this program could be written. If your program produces proper results, then you have been successful. One solution to the problem is shown in Figure 9-67. Compare your program with this one. If you were unsuccessful in writing a program, study Figure 9-67 very carefully until you understand the procedures involved.

Another problem that makes a good programming exercise is finding the square root of a number. Writing the program is not too difficult once you develop the proper algorithm. While there are many different ways to find the square root of a number, the easiest method from the programmer's point of view involves the subtraction of successive odd integers.

This method works because of the relationship between perfect squares. The first several perfect squares are  $0^2 = 0$ ,  $1^2 = 1$ ,  $2^2 = 4$ ,  $3^2 = 9$ ,  $4^2 = 16$ ,  $5^2 = 25$ , etc. Notice:

The relationship between the numbers 0, 1, 4, 9, 16, 25, etc.

The difference between 0 and 1 is 1, the first odd integer.

The difference between 1 and 4 is 3, the second odd integer.

The difference between 4 and 9 is 5, the third odd integer; etc.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	—	—	Product (LS byte)
0001	—	—	Product (MS byte)
0002	—	—	Multiplicand (LS byte)
0003	—	—	Multiplicand (MS byte)
0004	—	—	Multiplier
*	*	*	Instructions start at address 0010
0010	7F	CLR	Clear the product.
0011	00	00	
0012	00	00	
0013	7F	CLR	
0014	00	00	
0015	01	01	
0016	7D	TST	Test the multiplier.
0017	00	00	
0018	04	04	
0019	27	BEQ	If the multiplier is 0, branch to the WAI instruction.
001A	19	19	
001B	74	LSR	Otherwise, shift the right most bit of the multiplier into the C flag.
001C	00	00	
001D	04	04	
001E	24	BCC	If the C flag is 0 branch to here.
001F	0C	0C	
0020	96	LDAA	Otherwise, load the LS byte of the product into accumulator A.
0021	00	00	
0022	9B	ADDA	Then add the LS byte of the multiplicand.
0023	02	02	
0024	D6	LDAB	Load the MS byte of the product into accumulator B.
0025	01	01	
0026	D9	ADCB	Add (with carry) the MS byte of the multiplicand.
0027	03	03	
0028	97	STAA	Store the contents of accumulator A as the LS byte of the product.
0029	00	00	
002A	D7	STAB	Store the contents of accumulator B as the MS byte of the product.
002B	01	01	
002C	78	ASL	Shift the LS byte of the multiplicand to the left.
002D	00	00	
002E	02	02	
002F	79	ROL	Rotate the MS byte of the multiplicand to the left.
0030	00	00	
0031	03	03	
0032	20	BRA	Repeat the process.
0033	E2	E2	
0034	3E	WAI	Stop.

Figure 9-67

Program for multiplying a double-precision multiplicand by an 8-bit multiplier.

This relationship gives us a simple method of finding the exact square root of perfect squares and of approximating the square root of non-perfect squares.

The procedure for finding the square root of a number looks like this:

1. Subtract successive odd integers (1, 3, 5, 7, 9, etc.) from the number until the number is reduced to 0 or a negative value.
2. Count the number of subtractions required. The count is the exact square root of the number if the number was a perfect square. The count is the approximate square root if the number was not a perfect square.

For example, let's find the square root of  $49_{10}$ .

49	Original Number.
<u>-1</u>	Subtract the first odd integer.
48	
<u>-3</u>	Subtract the second odd integer.
45	
<u>-5</u>	Subtract the third odd integer.
40	
<u>-7</u>	Subtract the fourth odd integer.
33	
<u>-9</u>	Subtract the fifth odd integer.
24	
<u>-11</u>	Subtract the sixth odd integer.
13	
<u>-13</u>	Subtract the seventh odd integer.
0	Stop subtracting because the original number has been reduced to 0.

We simply count the number of subtractions required.

Since 7 subtractions were required, the square root of 49 is 7.

## Procedure (Continued)

9. With pencil and paper, use the above algorithm to find the square root of  $81_{10}$ . Does the answer give the exact square? \_\_\_\_\_. Was the result of the final subtraction 0? \_\_\_\_\_.
10. With pencil and paper, use the above algorithm to find the square root of  $119_{10}$ . How many subtractions are required to reduce the number to a negative value. Does this count approximate the square root of  $119_{10}$ ? \_\_\_\_\_.
11. Write a program that uses the above algorithm to find or approximate the square root of any unsigned 8-bit number.
12. Load your program into the Trainer and run it. Verify that it works for several different values.

## Discussion

Our solution to the problem is shown in Figure 9-68. The number is loaded into accumulator A, where it will be gradually reduced to a negative value. The odd integer is maintained in accumulator B. Each new odd integer is formed by incrementing twice. The SBA instruction is used to subtract the odd integer from the number. The BCS instruction is

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	96	LDAA	Load the number that is at this address into accumulator A. Load accumulator B with the first odd integer. Subtract the odd integer from the number.
0001	0F	0F	
0002	C6	LDAB#	
0003	01	01	
0004	10	SBA	If the carry is set, branch to here. Otherwise, form the next higher odd integer by incrementing B twice. Branch back to here. Shift the odd integer to the right. Store the answer at this address.
0005	25	BCS	
0006	04	04	
0007	5C	INCB	
0008	5C	INCB	Wait. Number to be operated upon. Final answer appears here.
0009	20	BRA	
000A	F9	F9	
000B	54	LSRB	
000C	D7	STAB	
000D	10	10	
000E	3E	WAI	
000F	—	Number	
0010	—	Answer	

Figure 9-68

Square root subroutine

used to determine when the number goes negative (a borrow occurs at that point). You could have used the BMI instruction but this would limit the original number to a value below  $+128_{10}$ . A few bytes are saved by not maintaining a separate count of the number of subtractions. Instead, the final odd integer value is converted to the count. This is possible because of the relationship between the odd integer value and the number of subtractions. As the program is written, the final odd integer is always one more than twice the number of subtractions. By shifting the final odd integer to the right, the correct count is created.

Of course, any square root program that is limited to numbers below  $256_{10}$  is of limited use. However, this same technique can be applied to multiple-precision numbers. Figure 9-69 shows a program that can find or approximate the square root of numbers up to  $16,385_{10}$ . Before you study this program, try to write your own program to do this.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	96	LDAA	Load accumulator A with the
0001	1A	1A	LS byte of the number.
0002	D6	LDAB	Load accumulator B with the
0003	19	19	MS byte of the number.
0004	7F	CLR	Clear
0005	00	00	the odd
0006	1B	1B	integer.
0007	7C	→ INC	Increment.
0008	00	00	the odd
0009	1B	1B	integer.
000A	90	SUBA	Subtract the odd
000B	1B	1B	integer from the LS byte of the
			number.
000C	C2	SBCB#	Take care of any borrow
000D	00	00	from the MS byte of the number.
000E	25	BCS	If the carry is set, branch
000F	05	05	to here.
0010	7C	→ INC	Otherwise, form the next
0011	00	00	higher odd integer by
0012	1B	1B	incrementing
0013	20	BRA	and branching
0014	F2	→ F2	to here.
0015	74	→ LSR	Convert the odd integer to
0016	00	00	the answer by shifting
0017	1B	1B	right.
0018	3E	WAI	Stop.
0019	—	Number (MS)	Number to be
001A	—	Number (LS)	operated upon.
001B	—	Odd integer	Form the odd integer and the
			answer here.

Figure 9-69

Routine for finding the square root of a double precision number.



## Experiment 9

### STACK OPERATIONS

#### OBJECTIVES:

*To demonstrate the stack operations that occur automatically.*

*To demonstrate ways that the programmer can use the stack.*

*To demonstrate the break-point capability of the Trainer.*

### Introduction

As you learned in Unit 6, the stack is used by the MPU to perform some automatic functions. When an interrupt occurs or a WAI is encountered, the MPU pushes the contents of the program counter, index register, accumulators, and condition codes on to the stack. We can easily verify this.

## Procedure

- Figure 9-70 shows a program for setting the MPU registers to a known state. Examine the program and determine the hex contents of the following registers immediately after the WAI is executed.

Condition Code Register \_\_\_\_\_  
 Accumulator B \_\_\_\_\_  
 Accumulator A \_\_\_\_\_  
 Index Register \_\_\_\_\_  
 Program Counter \_\_\_\_\_

- Load the program into the Trainer and verify that you loaded it properly.
- Execute the program using the DO command.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	8E	LDS#	Load 0020 into the stack pointer
0001	00	00	
0002	20	20	
0003	CE	LDX#	Load EEDD into the index register.
0004	EE	EE	
0005	DD	DD	
0006	C6	LDAB#	Load BB into ACCB.
0007	BB	BB	
0008	86	LDAA#	Load AA into ACCA.
0009	AA	AA	
000A	36	PSHA	Push AA onto the stack.
000B	86	LDAA#	Load CC into ACCA.
000C	CC	CC	
000D	06	TAP	Transfer CC into the condition codes.
000E	32	PULA	Pull AA from the stack.
000F	3E	WAI	Wait.
0010			

Figure 9-70

This routine sets the contents of all MPU registers to known values.

4. Examine the following memory locations and record their hex contents.

Address	Contents	Register
001A	_____	_____
001B	_____	_____
001C	_____	_____
001D	_____	_____
001E	_____	_____
001F	_____	_____
0020	_____	_____

5. Identify the register from which these numbers came.
6. Try to examine the contents of ACCA, ACCB, PC, SP, and INDEX register. Do their contents agree with the number loaded there?

## Discussion

When the WAI instruction is executed, the contents of the MPU registers are pushed onto the stack. Since the stack pointer is initially at 0020, the contents of the registers are stored as follows.

Address	Contents	Where it came from
001A	CC	Condition Codes
001B	BB	Accumulator B
001C	AA	Accumulator A
001D	EE	Index Register (high byte)
001E	DD	Index Register (low byte)
001F	00	Program Counter (high byte)
0020	10	Program Counter (low byte)

When you tried to examine the contents of ACCA, ACCB, SP, etc., you found that their contents did not agree with what was loaded. The reason for this **apparent** error is that the Trainer does not actually examine the contents of these registers. Instead, it examines what is placed in the stack by the WAI instruction. However, when the Trainer is reset, the monitor program assumes that the stack starts at address 00D1. Since our program moved the location of the stack, we can not use the ACCA, ACCB, PC, SP, CC, or INDEX commands after changing the stack pointer and then resetting the Trainer.

This demonstrates how the MPU uses the stack. A similar operation occurs for the SWI instruction or when a hardware interrupt occurs. Of course, the programmer can also use the stack.

### Procedure (Continued)

7. Figure 9-71 shows a program that will clear memory locations 0001 through 001F. It then transfers a list of numbers to these addresses. The numbers come from addresses 0151 through 016F.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ ADDRESS	COMMENTS
0020	CE	LDX #	Load the index register with highest address to be cleared. Clear it.
0021	00	00	
0022	1F	1F	
0023	6F	CLR, X	
0024	00	00	Decrement index register to next lower address. Finished? If not, go back and clear the indicated address. Set index register to first entry in new list.
0025	09	DEX	
0026	26	BNE	
0027	FB	FB	
0028	08	INX	Set the stack pointer to one less than the first entry in the old list. Pull the entry from the old list. Store it in the new list.
0029	8E	LDS #	
002A	01	01	
002B	50	50	
002C	32	PULA	Increment index register to next entry in list. Finished? If not, go back and pull next entry. Otherwise, wait.
002D	A7	STAA, X	
002E	00	00	
002F	08	INX	
0030	8C	CPX #	Finished? If not, go back and pull next entry. Otherwise, wait.
0031	00	00	
0032	20	20	
0033	26	BNE	
0034	F7	F7	
0035	3E	WAI	

Figure 9-71

Program for demonstrating stack operations and breakpoints.

8. Load this program into the Trainer and verify that you loaded it properly.
9. At address 0151 through 016F, load the numbers 01 through 1F<sub>16</sub>, respectively.
10. Execute the program using the DO command.
11. Examine addresses 0001 through 001F. They should contain the numbers 01 through 1F, respectively.

## Discussion

This illustrates how the stack can be used in conjunction with indexing to move a list of numbers.

When this program is executed using the DO command, everything happens so fast that it is impossible to see intermediate results. Of course, you could use the single-step mode and examine the result produced by every single instruction. But in many programs, this is a long, tedious process. Therefore, the Trainer provides another way to examine programs. It allows us to set four different breakpoints in our program. The Trainer will execute instructions at its normal speed until it reaches one of these breakpoints. At that point, the Trainer will stop with the address and op code of the next instruction displayed. While the Trainer is stopped, you can examine and change the contents of any register or memory location. When you are ready to resume, you depress the return (RTI) key and the Trainer executes instructions at its normal speed until the next breakpoint or a WAI instruction is encountered.

## Procedure (Continued)

12. Verify that the program is still in memory.
13. Depress the RESET key. Do not depress RESET again as you perform the following steps. To do so, will erase any breakpoints that you set.
14. Refer to the program listing in Figure 9-71. Let's assume we wish to stop and examine memory and the MPU registers just before the BNE instruction at address 0026 is executed.
15. Depress the BR key. The display should be \_ \_ \_ \_ Br. The Trainer is now ready to accept the first breakpoint address. Enter the address at which the Trainer is to stop: 0026. The breakpoint is now entered.
16. Without hitting RESET, depress the DO key. Enter the address of the first instruction in the program: 0020.
17. Immediately, the display will show the address 0026 and op code 26 at which the breakpoint occurred.
18. Without hitting RESET, examine the contents of the index register. It should now read 001E.
19. Depress the EXAM key and examine address 001F. It should now be cleared.
20. Notice that you can examine the contents of any MPU register or memory location from this breakpoint mode.
21. When you are ready for the program to resume, depress the RTI key once. Again, the display will read 002626 because the MPU is back at the same breakpoint on the second pass through the first loop.
22. Examine the index register again. It should now read 001D. Examine location 001E and verify that it has been cleared.

23. The loop will be repeated  $31_{10}$  times. On the  $32^{nd}$  pass, the program will escape the loop.
24. Before you go further, set a second breakpoint at the INX instruction. Do this by depressing the BR key and entering the address of the instruction (0028).
25. Depress the RTI key again. Notice that the program is still stopping at the first breakpoint. It will continue to do so until it escapes the first loop.
26. You have now pushed the RTI key three times. Repeatedly push the RTI key until the display changes to 0028 08. The RTI key should have been depressed a total of  $32_{10}$  times, counting the first three times.
27. The program is now waiting at the second break point.
28. To demonstrate a point, let's set two additional break points.
29. Depress the BR key and enter address 0029. This sets the third break point at the LDS# instruction.
30. Depress the BR key again and enter address 0033. This sets the fourth break point at the last BNE instruction.
31. The Trainer will accept only four breakpoints. We have now reached this limit. Depress the BR key again in an attempt to enter a fifth breakpoint. Notice that the word "FULL!" appears on the display.
32. Depress the RTI key so that the Trainer resumes program execution. It should stop at the third breakpoint.
33. Depress the RTI key again. The program should stop at the fourth breakpoint. Notice that the program is again in a loop. On each pass through the loop, the program will stop at this fourth breakpoint.
34. Analyze the operation of the program by examining the pertinent registers and memory locations on each pass through the loop.

## Discussion

The breakpoint capability of the Trainer can be a powerful aid in writing, analyzing and debugging a program. It allows us to stop at four distinct points in the program. Here are some tips to remember when using this capability:

1. A maximum of four breakpoints can be used.
2. These may be entered all at once or during a previous breakpoint pause.
3. The RESET key erases all breakpoints.
4. The contents of the address at which the breakpoint is set must be an op code.



## Experiment 10

### SUBROUTINES

#### OBJECTIVES:

To demonstrate the use of subroutines.

To demonstrate that the monitor program of the ET-3400 Trainer contains some useful subroutines that can be called when needed.

To gain experience writing programs.

#### Introduction

Most of the subroutines that you will develop and use in this experiment deal with lighting the displays on the Trainer. For this reason, we will begin by discussing how the displays are accessed.

The ET-3400 Microprocessor Trainer has six hexadecimal displays. Each display contains eight light-emitting diodes (LEDs) arranged as shown in Figure 9-72. Each LED is given two addresses. The addresses for the left-most display are shown. To light a particular LED, we simply store an odd number at the proper address. An odd number is used because the LED responds to a 1 in bit 0 of the byte that is stored. To turn an LED off, we store an even number at the proper address. The following procedure will demonstrate this.

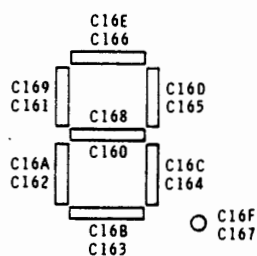


Figure 9-72

Addresses of the various segments in the left LED display.

## Procedure

1. Write a program that will halt after storing an odd number (such as 01) at address C167<sub>16</sub>.
2. Load the program into the Trainer and execute it using the DO command. The microprocessor should halt with the decimal point of the left-most display lit.
3. Notice that the LED remains lit until it is deliberately turned off.

## Discussion

To form characters, the LED's in the display must be turned on in combination. For example, to form the letter "A", the segments at addresses C162, C161, C166, C165, C164, and C160 must be turned on.

## Procedure (Continued)

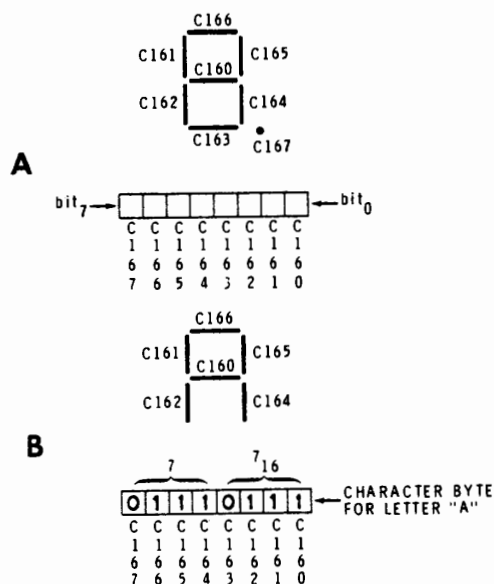
4. Write a program that will halt after storing an odd number (such as 01) at the six addresses listed above.
5. Load the program into the Trainer and execute it using the DO command. The microprocessor should halt with the letter A in the left-most display.

## Discussion

Your program probably took this form:

```
LDAA  #      01
STAA  C162
STAA  C161
STAA  C166
      .
      .
      .
WAI
```

While this approach works, the program would have to be rewritten for each new character. What is needed is a program that will form many characters. One approach is to store characters as 8-bit character bytes. Since there are eight LED's in each display, each bit of the character byte can be assigned to a different LED segment. Figure 9-73A shows how



**Figure 9-73**

### Assigning the bits of the character byte.

each bit in a character byte is assigned to each segment of the display. To light a corresponding LED, the proper bit in the character byte must be 1. For example, Figure 9-73B shows the character byte for the letter A. To form this letter, all display segments except C163 and C167 must be lit. Therefore, a 1 is placed in the character byte at all bits except the two that correspond to these addresses.

The display responds only to bit 0 of the character byte. To make each segment bit appear in turn at bit 0, the character byte must be shifted to the right. After each shift, the contents of the character byte must be stored at the address whose corresponding bit is now at bit 0. The procedure is:

1. Store the contents of the character byte at C160<sub>16</sub>.
2. Shift the character byte to the right.
3. Store it at C161<sub>16</sub>.
4. Shift it to the right again.
5. Store it at C162<sub>16</sub>.

Etc.

A program that will do this is shown in Figure 9-74.

### Procedure (Continued)

6. Load the program into the Trainer and verify that you loaded it correctly.
7. Execute the program using the DO command. The left-most digit should display the letter A.
8. The character byte is at address 0001. Change this byte to 47<sub>16</sub>.
9. Execute the program again using the DO command. What letter appears in the display? \_\_\_\_\_.
10. Change the character byte so that the letter H is displayed. What character byte is required? \_\_\_\_\_.

HEX ADDRESS	HEX CONTENTS	MNEMONIC/ CONTENTS	COMMENTS
0000	86	LDAA #	Load accumulator A immediate with the
0001	77	77	character byte.
0002	CE	LDX #	Load the index register immediate with
0003	C1	C1	the address.
0004	60	60	of the left display.
0005	A7	STAA, X	Store the character byte at the
0006	00	00	address indicated by the index register.
0007	44	LSRA	Shift the character bit to the right.
0008	08	INX	Advance index register to the address of the next segment.
0009	8C	CPX	Compare index register with one greater
000A	C1	C1	than the address of the
000B	68	68	last segment.
000C	26	BNE	If no match occurs branch
000D	F7	F7	back to here.
000E	3E	WAI	Otherwise, stop.

Figure 9-74

Program for lighting a display.

11. Change the character byte to  $79_{16}$ . Execute the program. What character is displayed? \_\_\_\_\_.
12. Refer to Figure 9-75. This figure shows the addresses of the LED's in each of the six displays. You have seen that the left display has an address of  $C16X_{16}$ . The X stands for some number between 0 and F, depending on which segment of that display we wish to use. The next display to the right has an address of  $C15X_{16}$ ; etc.
13. Now return to the program shown in Figure 9-74. Addresses 0003 and 0004 contain the address of the affected display. By changing this address, we can move the character to a different display. Actually since all display addresses start with C1, we need only change the number at address 0004.
14. Change the byte at 0004 to  $50_{16}$ . Change the byte at  $000B_{16}$  to 58. Execute the program using the DO command. The character should appear in the second display from the left.
15. Change the byte at 0004 to  $10_{16}$  and the byte at 000B to  $18_{16}$ . Execute the program using the DO command. The character should appear in the right-most display.

## Discussion

It has probably occurred to you that the monitor program must have a subroutine that performs this same function. Fortunately, this subroutine is written in such a way that we can use it. It is called OUTCH for OUTput CHAracter. It starts at address  $FE3A_{16}$ . We can call this subroutine anytime we like by using the JSR instruction. This subroutine assumes that the character byte is in accumulator A.

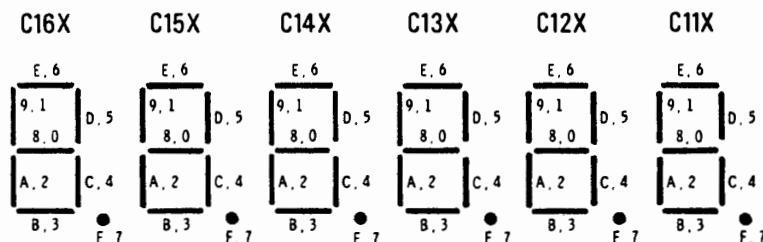


Figure 9-75

Addresses of the various display segments.

## Procedure

16. Load the program shown in Figure 9-76. Verify that you loaded it properly.
17. Execute the program using the DO command. What message does the program write? \_\_\_\_\_.
18. Notice that each character is written in a different display. Thus, the subroutine OUTCH automatically changes the address to that of the next display after each character is written.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	86	LDAA#	Load accumulator A immediate with the character byte for the letter H. Jump to subroutine OUTCH
0001	37	37	
0002	BD	JSR	
0003	FE	FE	
0004	3A	3A	Load ACCA with next character byte. Display it.
0005	86	LDAA#	
0006	4F	4F	
0007	BD	JSR	
0008	FE	FE	Load next character. Display it.
0009	3A	3A	
000A	86	LDAA#	
000B	0E	0E	
000C	BD	JSR	Load next character. Display it.
000D	FE	FE	
000E	3A	3A	
000F	86	LDAA#	
0010	67	67	Display it. Stop.
0011	BD	JSR	
0012	FE	FE	
0013	3A	3A	
0014	3E	WAI	

Figure 9-76

This program uses the OUTCH subroutine in the monitor program to display a message.

## Discussion

The monitor program writes several messages of its own. Examples are: ACCA, ACCB, CPU UP, and FULL! Thus, the monitor has a subroutine that can be used to write messages. It is called OUTSTR for OUTput a STRing of characters. Its starting address is at FE52<sub>16</sub>. There is a special convention for calling this subroutine. The JSR FE52<sub>16</sub> instruction must be followed immediately by the character bytes that make up the message. Up to six characters can be displayed. The last character must have the decimal point lit. After the message is displayed, control is returned to the instruction immediately following the last character.

## Procedure (Continued)

19. Load the program shown in Figure 9-77 into the Trainer and verify that you loaded it properly.
20. Execute the program using the DO command. What message does it display? \_\_\_\_\_.
21. Modify the program so that it displays HELLO.

HEX ADDRESS	HEX CONTENTS	MNEMONIC/ CONTENTS	COMMENTS
0000	BD	JSR	Jump to the subroutine that will display the following message.  H E L P. ← Decimal point must be lit in last character. Then stop.
0001	FE	FE	
0002	52	52	
0003	37	37	
0004	4F	4F	
0005	0E	0E	
0006	E7	E7	
0007	3E	WAI	

Figure 9-77

The OUTSTR subroutine in the monitor is used to display a message.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	BD	JSR	Cal OUTSTR.  N O. ← Decimal point lit (last character).
0001	FE	FE	
0002	52	52	
0003	76	76	
0004	FE	FE	
0005	BD	JSR	Call OUTSTR again.  G O. ← Decimal point lit (last character). Then stop.
0006	FE	FE	
0007	52	52	
0008	5E	5E	
0009	FE	FE	
000A	3E	WAI	

Figure 9-78  
OUTSTR is called twice.

22. The program shown in Figure 9-78 calls the OUTSTR subroutine twice. Load this program into the Trainer.
23. Execute it using the DO command. What message is displayed?  
\_\_\_\_\_.
24. Notice that the second message (GO.) is written to the right of the first. Thus, subroutine OUTSTR does not reset the display to the left for the second message.
25. Rewrite the program so that two blank displays appear between NO. and GO.



## Discussion

When displaying long messages such as: "HELLO CAN I HELP YOU?", the display must be given no more than six characters at a time. Also, a short delay must be placed between the various parts of the message. You can achieve a delay by loading the index register with FFFF and decrementing it to 0000. You can achieve an additional delay by using either accumulator in conjunction with the index register. We can write a display subroutine and call it between each part of the message.

Also, because we are using the same displays over again for each part of the message, each new word should start on the left. The subroutine called OUTSTR has an alternate entry point at address FD8C<sub>16</sub> called OUTSTJ. The calling convention for this subroutine is the same as that for OUTSTR. However, each new message starts in the left-most display.

## Procedure (Continued)

26. Load the program shown in Figure 9-79. Verify that you loaded it properly.
27. Execute the program using the DO command. What message is displayed? \_\_\_\_\_
28. Change the number in address 003C<sub>16</sub>, 003E<sub>16</sub>, and 003F<sub>16</sub>.
29. Execute the program using the DO command. What affect does this have?
30. Write a program of your own that will display "LOAD 2 IS BAD."

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	BD	JSR	Call Delay Subroutine
0001	00	00	
0002	3B	3B	
0003	BD	JSR	Call OUTSTJ
0004	FD	FD	
0005	8C	8C	
0006	37	37	H
0007	F	4F	E
0008	0E	0E	L
0009	0E	0E	L
000A	FE	FE	O.
000B	BD	JSR	Call Delay Subroutine
000C	00	00	
000D	3B	3B	
000E	BD	JSR	Call OUTSTJ again
000F	FD	FD	
0010	8C	8C	
0011	4E	4E	C
0012	77	77	A
0013	76	76	N
0014	00	00	blank
0015	B0	B0	I.
0016	BD	JSR	Call Delay Subroutine
0017	00	00	
0018	3B	3B	
0019	BD	JSR	Call OUTSTJ again
001A	FD	FD	
001B	8C	8C	
001C	37	37	H
001D	4F	4F	E
001E	0E	0E	L
001F	67	67	P
0020	80	80	•
0021	BD	JSR	Call Delay Subroutine
0022	00	00	
0023	3B	3B	
0024	BD	JSR	Call SUTSTJ again
0025	FD	FD	
0026	8C	8C	
0027	3B	3B	Y
0028	7E	7E	O
0029	3E	3E	U
002A	00	00	blank
002B	80	80	•
002C	BD	JSR	Call Delay Subroutine
002D	00	00	
002E	3B	3B	
002F	BD	JSR	Call OUTSTJ again
0030	FD	FD	
0031	8C	8C	
0032	00	00	blank
0033	00	00	blank
0034	00	00	blank
0035	00	00	blank
0036	00	00	blank
0037	80	80	•
0038	7E	JMP	Do it all again

- cont'd. -

-- cont'd. --			
HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0039	00	00	} Delay Subroutine
003A	00	00	
003B	86	LDA A#	
003C	02	02	
003D	CE	LDX#	
003E	00	00	
003F	00	00	
0040	09	DEX	
0041	26	BNE	
0042	FD	FD	
0043	4A	DECA	
0044	26	BNE	
0045	F7	F7	
0046	39	RTS	

Figure 9-79

This program makes extensive use of the subroutine call.

## Discussion

The monitor program in the Trainer contains some other useful subroutines. These are outlined in the manual for the ET-3400 Microprocessor Trainer. Two of the most useful are REDIS and OUTBYT.

OUTBYT is a subroutine that displays the contents of accumulator A as two hex digits. Its address is FE20<sub>16</sub>. When this subroutine is called for the first time, the two left displays are used. If it is called again without being reset, the two center displays are used. The third time, the two right displays are used.

The display can be reset to the left by calling the REDIS subroutine. This subroutine is located in address FCBC<sub>16</sub>. If OUTBYT is called after REDIS is called, the two left displays will be used.

## Procedure (Continued)

31. Load the program shown in Figure 9-80. Verify that you loaded it properly.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	4F	CLRA	Clear accumulator A
0001	BD	JSR	
0002	FE	FE	Call OUTBYT
0003	20	20	
0004	BD	JSR	
0005	00	00	Call Delay Subroutine
0006	0E	0E	
0007	4C	INCA	Increment accumulator A
0008	BD	JSR	
0009	FC	FC	Call REDIS
000A	BC	BC	
000B	7E	JMP	
000C	00	00	Do it again.
000D	01	01	
000E	CE	LDX#	
000F	FF	FF	
0010	FF	FF	
0011	09	DEX	
0012	26	BNE	
0013	FD	FD	
0014	39	RTS	

Figure 9-80

Using the OUTBYT and REDIS subroutines.

32. Execute the program using the DO command.
33. Which digits are used by the display? \_\_\_\_\_.
34. Notice that the JSR instruction at address 0008 calls the subroutine that resets the display to the left.
35. To illustrate why this is necessary, let's see what happens when this important step is omitted. Change the contents of locations 0008, 0009, and 000A to 01. This replaces the JSR instruction with three NOPs.

36. Execute the program using the DO command. Notice that, without calling the REDIS subroutine, the display advances to the right and is lost after the third time through the loop.
37. Restore the program to its original state. How can the count be speeded up?

## Discussion

The speed of the count can be varied by changing the contents of addresses 000F and 0010. It probably has occurred to you that the trainer could be turned into a digital clock. In the following procedure, you will develop a program that will do this.

## Procedure

38. Write a program that will count seconds from 00 to 99<sub>10</sub>. The seconds count should be maintained in the two left-most displays. It should count as the above program did, but in decimal instead of hexadecimal.
39. If you have problems, remember that the DAA instruction can be used to convert the addition of BCD numbers to a BCD sum. However, the DAA instruction works only if preceded immediately by an ADDA or ADCA instruction.
40. Load your program into the Trainer and execute it using the DO command.

## Discussion

One solution is shown in Figure 9-81. Carefully study this program. This routine counts the seconds in decimal. However in a real digital clock, the seconds reset to 00 after 59<sub>10</sub> rather than after 99<sub>10</sub>.

There are two one-second delay sub-routines listed in the following experiments. You must use the one that matches the clock frequency of your ET-3400 Trainer.

The original Trainer has a clock frequency of approximately 500 kHz. If your Trainer has not been modified, you must use the "Slow Clock One-Second Delay Subroutine."

If your Trainer has been modified for use with the Heathkit Memory I/O Accessory ETA-3400, it has a clock frequency of 1 MHz. In this case, you must use the "Fast Clock One-Second Delay Subroutine."

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	4F	CLRA	Clear seconds.
0001	BD	JSR	
0002	FE	FE	Call OUTBYT
0003	20	20	
0004	BD	JSR	
0005	00	00	Call Delay subroutine
0006	10	10	
0007	8B	ADDA#	Increment seconds
0008	01	01	
0009	19	DAA	Make it decimal
000A	BD	JSR	
000B	FC	FC	Call REDIS
000C	BC	BC	
000D	7E	JMP	
000E	00	00	Do it all again.
000F	01	01	
* 0010	CE	LDX#	} Slow Clock One-Second Delay Subroutine
0011	C5	C5	
0012	00	00	
0013	09	DEX	
0014	26	BNE	
0015	FD	FD	
0016	39	RTS	
cont'd			

• cont'd. •			
HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0010	36	PSHA	{ Fast Clock One-Second Delay Subroutine
0011	86	LDAA#	
0012	02	02	
0013	CE	LDX#	
0014	F3	F3	
0015	80	80	
0016	09	DEX	
0017	26	BNE	
0018	FD	FD	
0019	4A	DECA	
001A	26	BNE	
001B	F7	F7	
001C	32	PULA	
001D	39	RTS	

\*Use either the Fast Clock or the Slow Clock One-Second Delay Subroutine.

Figure 9-81

This routine counts seconds from 00 to 99.

## Procedure (Continued)

41. Modify your program (or the one in this Experiment) so that it displays seconds from 00 to 59 and then returns to 00 and starts over again.
42. Load your program into the Trainer and execute it using the DO command.
43. Debug your program if necessary until it performs properly.

## Discussion

One solution is shown in Figure 9-82. The seconds count is compared to 60 each time it is incremented. When it reaches 60, it is reset to 00.

The next step is to add a minutes count. This can be done by incrementing a decimal number each time the seconds count "rolls over" from 59 to 00. The decimal number is then displayed as minutes.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	C6	LDAB#	Load number for comparison
0001	60	60	
0002	4F	CLRA	Clear seconds.
0003	BD	JSR	
0004	FE	FE	Call OUTBYT
0005	20	20	
0006	BD	JSR	
0007	00	00	Call Delay Subroutine
0008	14	14	
0009	BD	JSR	
000A	FC	FC	Call REDIS
000B	BC	BC	
000C	8B	ADDA#	Increment seconds.
000D	01	01	
000E	19	DAA	Make it decimal
000F	11	CBA	Time to clear seconds
0010	27	BEQ	Yes.
0011	F0	F0	
0012	20	BRA	No.
0013	EF	EF	
* 0014	CE	LDX#	} Slow Clock One-Second Delay Subroutine
0015	C5	C5	
0016	00	00	
0017	09	DEX	
0018	26	BNE	
0019	FD	FD	
001A	39	RTS	
0014	36	PSHA	} Fast Clock One-Second Delay Subroutine
0015	86	LDAA#	
0016	02	02	
0017	CE	LDX#	
0018	F3	F3	
0019	80	80	
001A	09	DEX	
001B	26	BNE	
001C	FD	FD	
001D	4A	DECA	
001E	26	BNE	
001F	F7	F7	
0020	32	PULA	
0021	39	RTS	

\*Use either the Fast Clock or the Slow Clock One-Second Delay Subroutine.

Figure 9-82

This routine counts seconds from 00 to 59.



## Procedure (Continued)

44. Write a program that will display minutes and seconds properly. The minutes should be displayed in the two left displays; the seconds in the two center displays. Like the seconds, the minutes should return to 00 after 59.
45. Load your program and execute it.
46. Debug your program as necessary.

## Discussion

A solution is shown in Figure 9-83. Your approach may be more straightforward, but may require more memory.

The final step is to include the hours display.

## Procedure (Continued)

47. Modify your program so that it displays hours, minutes and seconds.
48. Load your program and execute it.
49. Debug your program as necessary.

A solution is shown in Figure 9-84. This program evolved over a period of time and is extremely compact. It is virtually impossible for a beginning programmer to write a program this compact on the first try. Your program may require substantially more memory, but the important thing is: does it work?

While you can "fine tune" the slow-clock period by changing the numbers in addresses 0004 and 0005, the clock will never be very accurate because it is temperature sensitive. The fast clock period is much more accurate because the oscillator is crystal controlled. You can fine tune it by changing the numbers in addresses 003A and 003B. In a later experiment, you will rectify this problem and produce an extremely accurate clock.

HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	00	00	Reserved for seconds
0001	00	00	Reserved for minutes
* 0002	CE (36)	LDX# (PSHA)	Slow Clock (call Fast Clock One-Second One-Second delay. Delay)
0003	C5 (BD)	C5 (JSR)	
0004	00 (00)	00 (00)	
0005	09 (2F)	DEX (2F)	
0006	26 (32)	BNE (PULA)	
0007	FD (01)	FD (NOP)	Load number for comparison.
0008	C6	LDAB#	
0009	60	60	Set carry bit.
000A	0D	SEC	
000B	8D	BSR	Branch to subroutine to increment seconds.
000C	11	11	
000D	8D	BSR	Branch to the same subroutine to increment minutes.
000E	0F	0F	
000F	BD	JSR	Call REDIS
0010	FC	FC	
0011	BC	BC	Load minutes
0012	96	LDAA	
0013	01	01	Call OUTBYT to display minutes.
0014	BD	JSR	
0015	FE	FE	Load seconds
0016	20	20	
0017	96	LDAA	Call OUTBYT to display seconds
0018	00	00	
0019	BD	JSR	Do it all again.
001A	FE	FE	
001B	20	20	Load seconds (or minutes) into A.
001C	20	BRA	
001D	E4	E4	Increment if necessary
001E	A6	LDAA, X	
001F	00	00	Adjust to decimal
0020	89	ADCA#	
0021	00	00	Time to clear?
0022	19	DAA	
0023	11	CBA	No.
0024	26	BNE	
0025	01	01	Yes.
0026	4F	CLRA	
0027	A7	STAA, X	Store seconds (or minutes)
0028	00	00	
0029	08	INX	Complement carry bit
002A	07	TPA	
002B	88	EORA#	Increment subroutine
002C	01	01	
002D	06	TAP	
002E	39	RTS	

cont'd.-

cont'd.			
002F	(86)	(LDAA #)	Fast Clock One-Second Delay Subroutine
0030	(02)	(02)	
0031	(CE)	(LDX #)	
0032	(F3)	(F3)	
0033	(80)	(80)	
0034	(09)	(DEX)	
0035	(26)	(BNE)	
0036	(FD)	(FD)	
0037	(4A)	(DECA)	
0038	(26)	(BNE)	
0039	(F7)	(F7)	
003A	(39)	(RTS)	

\*Numbers in parenthesis are for Fast Clock One-Second Delay only.

**Figure 9-83**

Routine for displaying minutes and seconds.

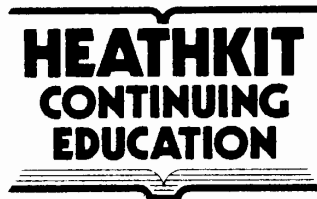
HEX ADDRESS	HEX CONTENTS	MNEMONICS/ CONTENTS	COMMENTS
0000	00	00	Reserved for seconds
0001	00	00	Reserved for minutes
0002	00	00	Reserved for hours
* 0003	CE (36)	LDX# (PSHA)	Slow Clock (Call Fast Clock One-Second One-Second Delay Delay)
0004	C5 (BD)	C5 (JSR)	
0005	00 (00)	00 (00)	
0006	09 (37)	DEX (37)	
0007	26 (32)	BNE (PULA)	
0008	FD (01)	FD (NOP)	Minutes and seconds will be compared with sixty. Prepare to increment seconds Go to subroutine that will increment seconds. Go to same subroutine. It will increment
0009	C6	LDAB#	
000A	60	60	
000B	0D	SEC	
000C	8D	BSR	
000D	11	11	Minutes if necessary. Hours will be compared with twelve. Go to same subroutine. It will increment hours if necessary.
000E	8D	BSR	
000F	0F	0F	
0010	C6	LDAB#	
0011	12	12	
0012	8D	BSR	Call REDIS Call display subroutine to display hours. Call display subroutine to display minutes. Call display subroutine to display seconds. Do it all again.
0013	0B	0B	
0014	BD	JSR	
0015	FC	FC	
0016	BC	BC	
0017	8D	BSR	Load seconds (or minutes or hours). Increment if necessary. Adjust to decimal. Time to clear? No.
0018	17	17	
0019	8D	BSR	
001A	15	15	
001B	8D	BSR	
001C	13	13	Yes. Store seconds (or minutes or hours). Point index register at minutes (or hours). Complement carry bit
001D	20	BRA	
001E	E4	E4	
001F	A6	LDAA, X	
0020	00	00	
0021	89	ADCA#	Point index register at hours (or minutes or seconds) Load hours (or minutes or seconds) Display hours (or minutes or seconds)
0022	00	00	
0023	19	DAA	
0024	11	CSA	
0025	25	BCS	
0026	01	01	Increment Subroutine
0027	4F	CLRA	
0028	A7	STAA, X	
0029	00	00	
002A	08	INX	
002B	07	TPA	Display Subroutine
002C	88	EORA#	
002D	01	01	
002E	06	TAP	
002F	39	RTS	
0030	09	DEX	Display Subroutine
0031	A6	LDAA, X	
0032	00	00	
0033	7E	JSR	
0034	FE	FE	
0035	20	20	Display Subroutine
0036	39	RTS	

0037	(86)	(I.DAA #)	Fast Clock One-Second Delay Subroutine
0038	(02)	(02)	
0039	(CE)	(I.DX #)	
003A	(F3)	(F3)	
003B	(80)	(80)	
003C	(09)	(DEX)	
003D	(26)	(BNE)	
003E	(FD)	(FD)	
003F	(4A)	(DECA)	
0040	(26)	(BNE)	
0041	(F7)	(F7)	
0042	(39)	(RTS)	

\*Numbers in parentheses are for Fast Clock One-Second Delay only.

Figure 9-84  
Twelve-hour clock program





# Individual Learning Program

## MICROPROCESSORS

*Unit 10*

### INTERFACING EXPERIMENTS

EE-3401

HEATH COMPANY  
BENTON HARBOR, MICHIGAN 49022

Copyright © 1977  
Heath Company  
All Rights Reserved  
Printed in the United States of America

## CONTENTS

Introduction .....	10-3
Experiment 1 Memory Circuits .....	10-4
Experiment 2 Clock .....	10-19
Experiment 3 Address Decoding .....	10-25
Experiment 4 Data Output .....	10-39
Experiment 5 Data Input .....	10-53
Experiment 6 Introduction To The Peripheral Interface Adapter (PIA) .....	10-65
Experiment 7 Audio Output .....	10-71
Experiment 8 Key Matrix And Parallel-To-Serial Conversion .....	10-83
Experiment 9 Digital-To-Analog And Analog-To- Digital Conversion .....	10-96
Schematic .....	10-111



## Unit 10

# INTERFACING EXPERIMENTS

### INTRODUCTION

This Unit contains nine interfacing experiments that are to be assembled and run on the Microprocessor Trainer. Most of the circuit parts for these experiments are supplied with this course. The remaining parts were part of the Trainer Kit.

You will be instructed to perform these experiments at the end of Units 7 and 8. Do not confuse them with the programming experiments in Unit 9. When you complete an experiment, you will be directed to the next experiment, or back to the Unit Activity Guide of the unit that directed you to the experiment.

If your Trainer is Model Number ET-3400, and has been modified for use with the Heathkit Memory I/O Accessory, Model ETA-3400, disconnect the 40-pin plug that connects the Trainer to the Memory I/O Accessory. Then reinstall the 2112 RAM IC's at IC-14 through IC-17 before starting the experiments in this unit.

If your Trainer is model number ET-3400A, and has been modified for use with the Heathkit Memory I/O Accessory, disconnect the 40-pin plug that connects the Trainer to the Memory I/O Accessory. Then reinstall the 2114 RAM IC's at IC14 and IC15 before starting the experiments in this unit.

## Experiment 1

### MEMORY CIRCUITS

#### OBJECTIVES:

*Show how memory circuits can be connected to a microprocessor.*

*Demonstrate timing requirements when using memory circuits.*

*Show how data is stored and read from memory circuits.*

*Demonstrate an elementary memory test to ensure proper, reliable operation.*

#### Introduction

In this experiment, you will construct a memory on the large connector block of the Microprocessor Trainer and interface the circuit with the Trainer circuits.

The initial sections of the experiment will examine memory and its characteristics. The remaining experiment section will interface the memory with the microprocessor and its support circuits. This program and all remaining hardware experiment programs will use a computer print-out listing. A detailed explanation of how to read the listing will be given later in the experiment.

#### TRAINER POWER REVIEW

With the Trainer plugged in and the Power switch off, the display LED's and the +5, +12, and -12 volt connector blocks are disconnected from Trainer power. The single LED next to the Power switch indicates this condition. Whenever you make connections between the Trainer and the large connector block, **always** switch the power off. This will not disturb any program stored in the Trainer. If you must remove or install a component in the Trainer circuits, such as an IC, remove the power plug from the wall receptacle.

## Material Required

- 1 Microprocessor Trainer
  - 1 1000 ohm, 1/4-watt, 10% resistor
  - 1 Pushbutton switch (#1)
  - 1 7400 integrated circuit (443-1)
  - 1 74126 integrated circuit (443-717)
  - 1 74LS30 integrated circuit (443-732)
  - 1 74LS27 integrated circuit (443-800)
  - Hookup wire (22 gauge, solid)
  - 1 IC puller tool
  - Hookup wire (22 gauge, solid)
- } From Trainer  
Parts Package

### ADDITIONAL MATERIAL REQUIRED (TRAINER ET-3400A)

- 2 2112-2 IC's (Heath Number 443-721)

## Procedure

1. Turn the Trainer power off, then unplug your Trainer from its wall receptacle.
2. Insert the 74126 (443-717) and 7400 (443-1) integrated circuits (IC's) into the large connector block as shown in Figure 10-1. Always install an IC in the block with pin 1 toward the left.

NOTE: If you are using Trainer model number ET-3400, perform step 3. If your Trainer is an ET-3400A, perform step 3A. All other steps of this procedure are common to both Trainer types, except where indicated.

3. Using the IC puller tool, remove the 2112 (443-721) IC from its socket at location IC17. (Observe the precautions described in Unit 9 for MOS devices.) Then insert the IC into the large connector block as shown in Figure 10-1. This IC will be reinstalled in the ET-3400 Trainer in a later experiment.
- 3A. Locate a 2112-2 IC (Heath number 443-721) that was supplied with this course. Notice that this IC is packed in conductive foam.

NOTE: These IC's are rugged, reliable components. However, normal static electricity discharged from your body through an IC pin to an object can damage the IC. Install these IC's without interruption as follows:

- Remove the IC from its package with both hands.
- Hold the IC with one hand and straighten any bent pins with the other hand.
- Insert the IC into the large connector block as shown in Figure 10-1.

4. Install the pushbutton switch at the location shown in Figure 10-1. Be sure to press straight down when you insert the switch leads — they are fragile.
5. Using the 22 gauge, solid hookup wire, interconnect the IC's and switch as shown in Figure 10-1. Install the 1000 ohm, 1/4-watt, 10% resistor at this time also.

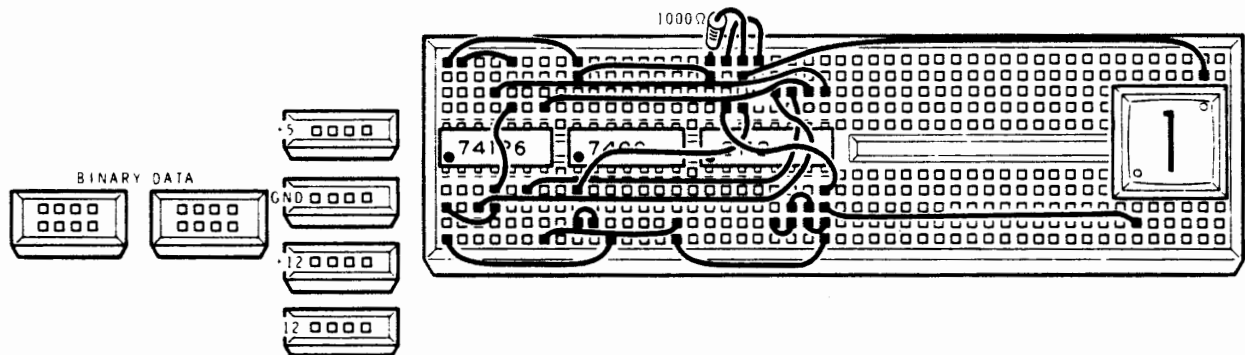


Figure 10-1

Part A of wire interconnect diagram.



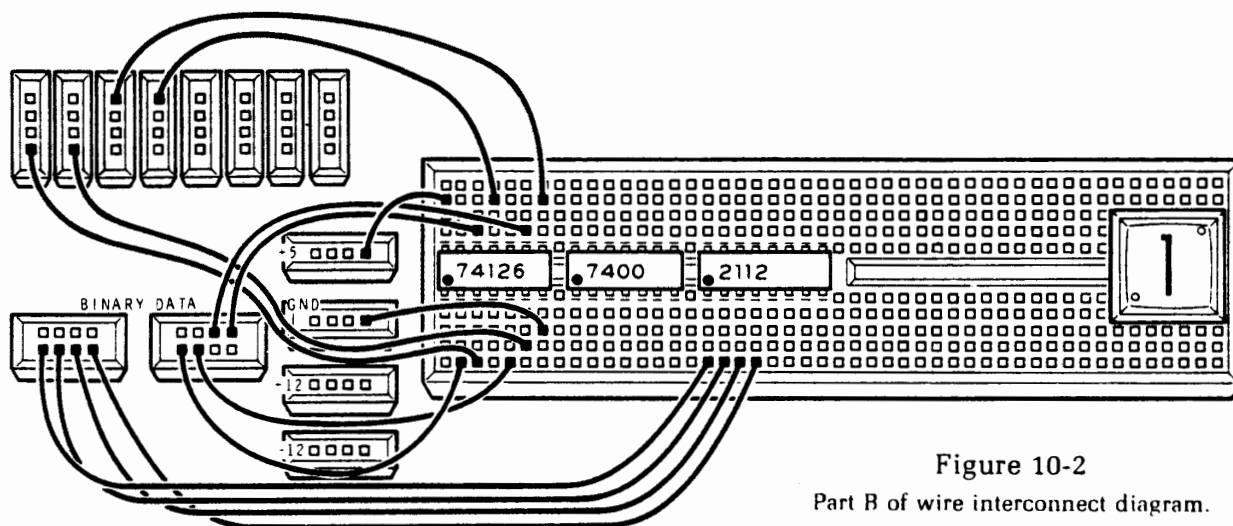


Figure 10-2

Part B of wire interconnect diagram.

6. Refer to Figure 10-2 and install hookup wire as shown. Now compare your circuit with the circuit shown in Figure 10-3. Figures 10-1 and 10-2 are supplied to familiarize you with the proper wiring technique. The remaining hardware experiments will show only the circuit diagram.
7. Connect your Trainer line cord plug to a wall receptacle, and switch the circuit power on. The four data LED's may or may not be randomly lit.

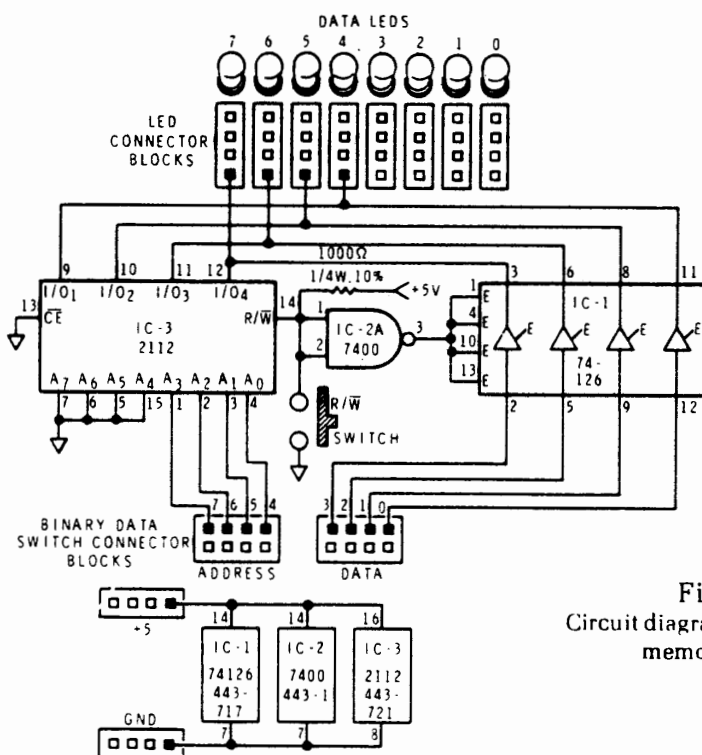


Figure 10-3

Circuit diagram of the first part of the memory experiment.

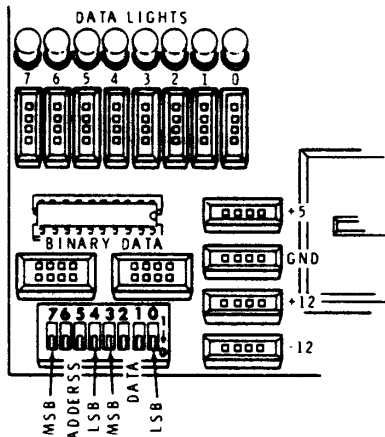


Figure 10-4

Binary data switch functions.

NOTE: The slide switch assembly in the lower left corner of the Trainer (Figure 10-4) is used to control address and data information in this experiment. The first four switches (0 thru 3) control data, while the next four (4 thru 7) control the address. Each group is arranged in a binary sequence with the least significant bits at 0 (data) and 4 (address). The physical position of each switch indicates a logic level; up for a logic 1 and down for a logic 0.

The pushbutton switch mounted on the large connector block functions as the memory Read/Write switch. In its out (off) position, the memory is in the read mode, and the four data LED's will display data stored in memory. When the R/W switch is pressed, the memory goes to the write mode, and the value stored in the four data switches is read into memory. The four data LED's immediately display the new memory data.

8. Set the four address switches to  $0000_2$ , and the four data switches to  $0000_2$ . Then press the R/W switch. If any of the data LED's were previously lit, they should now be out. This indicates that  $0000_2$  is now stored at address  $0000_2$ .
9. Now select address  $0001_2$  and set the data switches to  $0001_2$ . The display will show a random value produced at power-on. Press the R/W switch. The data lights will show  $0001_2$ , which is now stored in memory at address  $0001_2$ .
10. Refer to Figure 10-5 and use the slide switch assembly to load the remaining  $14_{10}$  address locations (2 thru  $15_{10}$ ) with the data specified. NOTE: To write data in memory; select the address, select the data value, and load the data by pressing the R/W switch.
11. With the address switches, select memory location  $9_{16}$ . The displayed data is  $\_\_\_\_2$ . Since the 16 memory locations contain a data value that matches the address, the displayed value should equal  $9_{16}$ . Randomly select various memory locations. As each location is selected, the stored data will be displayed.

ADDRESS		DATA	
HEX	BINARY	BINARY	
0	0000	0000	
1	0001	0001	
2	0010	0010	
3	0011	0011	
4	0100	0100	
5	0101	0101	
6	0110	0110	
7	0111	0111	
8	1000	1000	
9	1001	1001	
A	1010	1010	
B	1011	1011	
C	1100	1100	
D	1101	1101	
E	1110	1110	
F	1111	1111	

Figure 10-5

First data table for memory storage experiment.



12. Refer to Figure 10-6 and enter the specified data for each address location.
13. Randomly select a number of memory locations and note the stored data. You probably recognized the relationship between address and data while you entered the data. If not, do you now? The data corresponds to the 1's complement of the address. This circuit is being used as a look-up table. By selecting an address, you can retrieve data previously stored away. Similarly, this circuit can be used for code conversion. Using a binary sequence (as in Figures 10-5 and 10-6) for addressing, a value code can be stored to represent the address (for example, the Gray code as described in Unit 1). Thus, to find the Gray code value of  $8_{16}$ , simply examine memory location  $8_{16}$ . A second memory circuit can then be used to reconvert the code by using the Gray code values (in this example) for address locations, and the binary values for data.

ADDRESS		DATA
HEX	BINARY	BINARY
0	0000	1111
1	0001	1110
2	0010	1101
3	0011	1100
4	0100	1011
5	0101	1010
6	0110	1001
7	0111	1000
8	1000	0111
9	1001	0110
A	1010	0101
B	1011	0100
C	1100	0011
D	1101	0010
E	1110	0001
F	1111	0000

Figure 10-6  
Second data table for memory storage  
experiment.

## Discussion

In this section of the experiment, you used a  $256 \times 4$ -bit RAM integrated circuit. An IC of this type will have eight address pins and four I/O (input/output) pins through which a 4-bit data word may be stored (written) or read. The direction of I/O flow is determined by the  $R/\overline{W}$  (read/write) pin logic level. A logic 1 on this pin defines the four I/O pins as outputs, thus placing the IC in its **read** mode. A logic 0 on the  $R/\overline{W}$  pin defines the four I/O pins as inputs, thus placing the IC in its **write** mode. A chip enable (CE) pin allows the IC to be enabled or disabled without disturbing its memory contents. This feature will be more meaningful in a later experiment.

You also used a 3-state buffer array (four buffers) with the memory circuit. This is necessary to isolate the data switches when the memory is in its read mode. Each buffer acts as an open circuit at its output pin unless a logic 1 is applied to each enable (E) pin. As shown in Figure 10-3, when the  $R/\overline{W}$  switch is pressed, the memory  $R/\overline{W}$  line goes low (write mode), and the enable lines to the buffers go high (through NAND gate IC-2A). Switch data is coupled through the buffers and is written into memory. The LED's display the data value. When the  $R/\overline{W}$  switch is released, the memory returns to its read mode and the buffers return to their open circuit condition. The LED's now display the data value stored in memory.

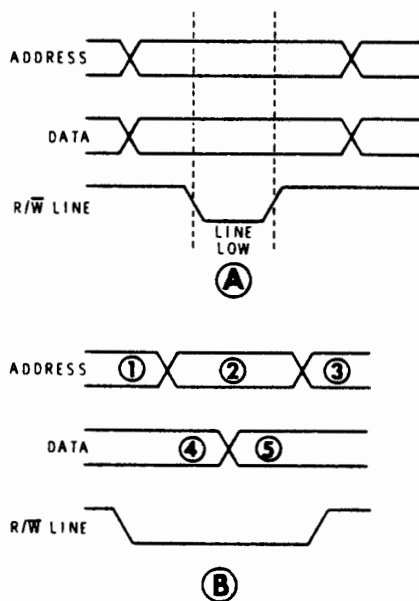


Figure 10-7  
Memory write timing diagram.

Writing into memory requires careful timing. Normally the address and data lines must be stable while the  $R/\bar{W}$  line is pulsed low, as shown in Figure 10-7, part A. Actually, the data is stored as soon as the  $R/\bar{W}$  line reaches a logic 0 level. If the data changes value while the  $R/\bar{W}$  line is low, the memory will store the new data. In like manner, if the address changes with the  $R/\bar{W}$  line low, the same data will be stored at the new address.

Figure 10-7 part B shows an extreme example of **improper** timing. At address condition 1, data 4 will be stored. Then data 4 will be stored at address 2. However, while at address 2, data changes to condition 5. Therefore, data 5 is now stored at address 2. Finally, data 5 is stored at address condition 3.

### Procedure (continued)

14. Turn the Trainer power off, then pull the power plug from the wall receptacle.

NOTE: If you are using Trainer model number ET-3400, perform step 15. For model number ET-3400A, perform step 15A. All other steps of this procedure are common to both trainers, except where indicated.

15. Using the IC puller tool, remove memory IC2112 (443-721) from its socket at location IC16. (Observe the necessary precautions for handling MOS devices.) Then insert the IC into the large connector block, next to the other memory IC.
- 15A. Locate the second IC 2112 (443-721) that was supplied with this course. (Observe the necessary precautions for handling MOS devices.) Then insert the IC into the large connector block, next to the other memory IC.

- 16A. Using hookup wire, rewire the connector block IC's and Trainer to form the circuit shown in Figure 10-8. Notice that most of the circuitry is identical to the first circuit you built. You may find it helpful to trace each on the Figure with a red pencil, as you install the wire.
- 16B. Connect your Trainer line cord plug to a wall receptacle, and switch the circuit power on.
17. The address, data, and  $R/\overline{W}$  switches function as before. Refer to Figure 10-9 and enter the data shown, beginning at address 0000<sub>2</sub>. NOTE: Data LED0 will be lit for the first eight address locations. Data LED1 will be lit for the last eight address locations.
18. Data LED's 0 and 1 indicate which memory IC is enabled. LED0 lights for IC4 and LED1 lights for IC3. Examine a number of addresses between 0000<sub>2</sub> and 0111<sub>2</sub>. Notice which memory IC is enabled. The data stored should match the address. Now examine a number of addresses between 1000<sub>2</sub> and 1111<sub>2</sub>. Notice which memory IC is enabled. The data stored should be the 1's complement of the address.
19. Switch the Trainer power off for a few seconds, then switch it back on. Examine a number of memory locations. Have their contents been altered? Do data LED's 0 and 1 still indicate which memory IC is enabled?



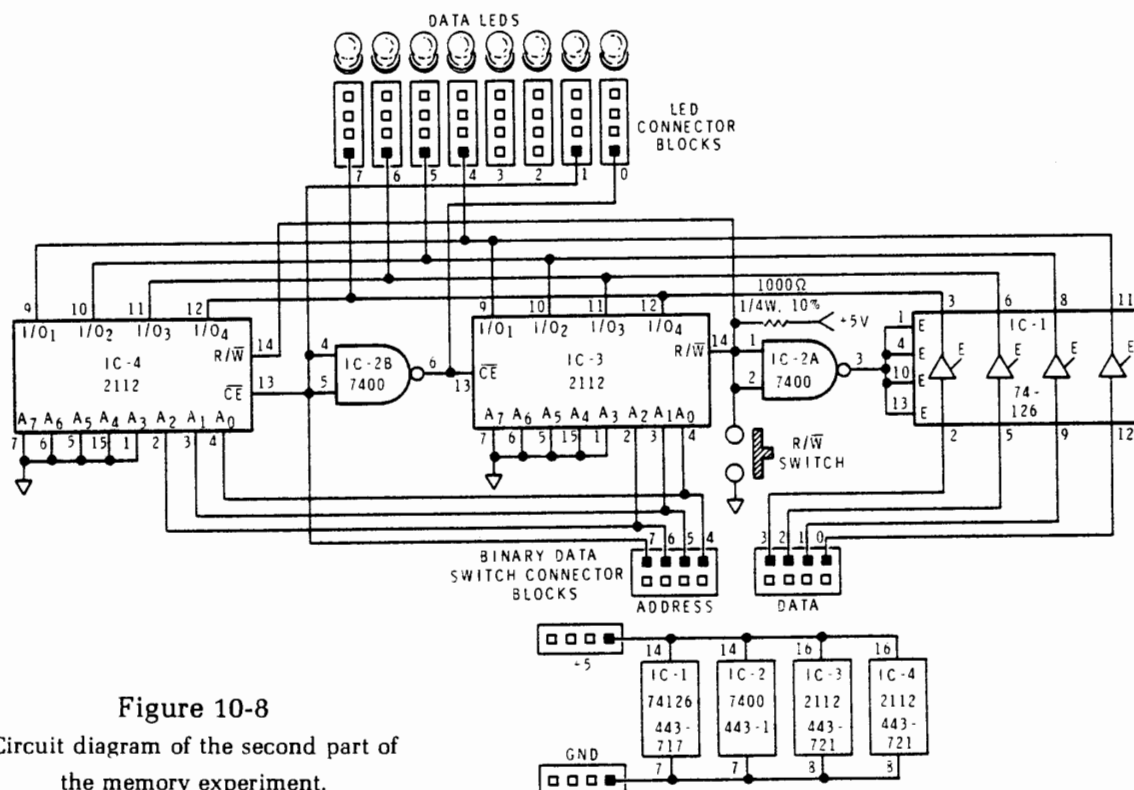


Figure 10-8

Circuit diagram of the second part of the memory experiment.

## Discussion

Refer to Figure 10-8 and note how data LED's 0 and 1 are connected. They indicate which memory IC is enabled by the MSB address switch. Since IC3 and IC4 share address lines 4, 5, and 6, the chip enable ( $\overline{CE}$ ) pins determine which IC is enabled for data transfer. The most significant address line is connected to pin 13 of IC4 and its complement is connected to pin 13 of IC3. Therefore, when bit 4 (switch 7) of the address is logic 0, IC4 is enabled; and when bit 4 is logic 1, IC3 is enabled.

When either memory IC is disabled, through pin  $\overline{CE}$ , stored data remains unaffected. The 3-state output pins go to an open circuit condition. This insures that only one memory IC on the common address lines will be active for data transfer.

RAM, unlike ROM, has a volatile memory. Therefore, when power is lost (even momentarily) data stored in memory is no longer valid. However, data can be reentered into memory and retained as long as power remains on.

ADDRESS HEX	BINARY	DATA BINARY
0	0000	0000
1	0001	0001
2	0010	0010
3	0011	0011
4	0100	0100
5	0101	0101
6	0110	0110
7	0111	0111
8	1000	0111
9	1001	0110
A	1010	0101
B	1011	0100
C	1100	0011
D	1101	0010
E	1110	0001
F	1111	0000

Figure 10-9

Third data table for memory storage experiment.

## Procedure (continued)

20. Switch the Trainer power off.
21. Remove the hookup wires (save them), pushbutton switch, resistor, and IC1 and IC2, used in the previous experiment, from the Trainer.
22. Refer to Figure 10-10, install the IC's, and wire the circuit to the Trainer. Caution: When you install the IC's, leave an extra set of holes between IC2 and IC3. You will be replacing the 14-pin IC2 with a 16-pin IC at a later time.
23. The memory circuit you have wired now interfaces with the microprocessor and will allow data transfer from address  $0200_{16}$  through  $02FF_{16}$ . Use the Trainer Examine function and randomly select an address in this memory block. Change the data at this location to  $AA_{16}$ . Press the FWD key, then press the BACK key. Is the memory content still  $AA_{16}$ ?

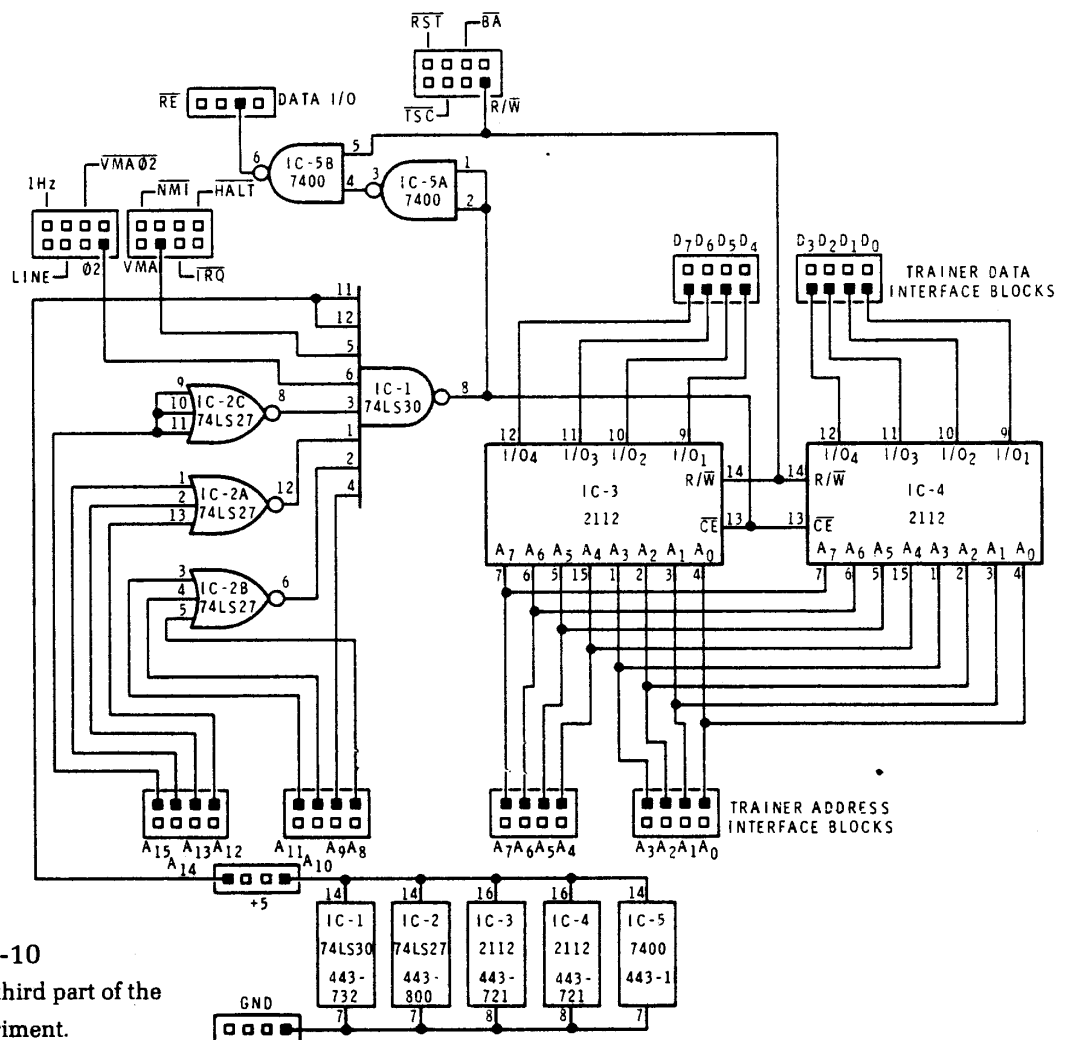


Figure 10-10  
Circuit diagram of the third part of the  
memory experiment.

24. Examine address  $0300_{16}$ . Now change the contents to  $AA_{16}$ . Press the FWD key, then press the BACK key. Is the memory content still  $AA_{16}$ .

The data you entered in the previous step was retained because of the memory circuit you wired into the Trainer. The data at location  $0300_{16}$  was not retained because no memory exists for that location.

25. Examine address  $0200_{16}$  and change its contents to  $AA_{16}$ .
26. Without switching the Trainer power off, interchange the  $D_6$  and  $D_7$  wires at the Data Interface Block, as shown in Figure 10-11.
27. Press the FWD key, then press the BACK key. The indicated data is now  $6A_{16}$ . By interchanging the sixth and seventh data bits,  $1010\ 1010_2$  became  $0110\ 1010_2$ . The memory IC still retains the original data you programmed. To see this relationship, return the two wires to their original position. Since the display has "latched in" the previous data, press the FWD key and then the BACK key. The correct memory contents are now displayed.

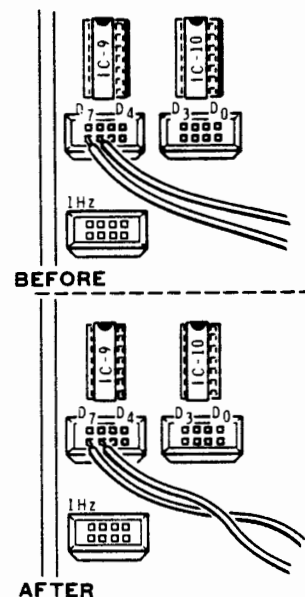


Figure 10-11

Physical manipulation of data by interchanging data lines.

## Discussion

IC1 and IC2 shown in Figure 10-10, form the address decoder. The inputs of this decoder are connected to address lines  $A_8$  thru  $A_{15}$ ,  $\phi_2$ , and VMA.

To enable the two memory IC's, the  $\overline{CE}$  pins must be at logic 0. This will occur when all of the inputs to IC1 are at logic 1. Therefore, only address  $0000\ 0010_2$  ( $02_{16}$ ) will decode properly. This is the high order byte of a 16-bit address.  $\phi_2$  and VMA will go to a logic 1 sometime during a proper address cycle. When this occurs, the output of IC1 will go to a logic 0, and memory will be enabled.

The low order byte of the 16-bit address determines the memory location in IC3 and IC4 where data will be stored or retrieved. Since each memory IC can store only four bits of data, two IC's are connected in parallel. Thus,  $256_{10}$  8-bit data words can be stored from address  $0200_{16}$  thru  $02FF_{16}$ . Therefore, in the circuit shown in Figure 10-10, address bits  $A_0$  thru  $A_7$  select the memory location and address bits  $A_8$  thru  $A_{15}$  select the specific memory IC's to be enabled.

Data flow direction is determined by the  $R/\overline{W}$  line. When this line is at logic 0, the MPU can write into memory. When this line is at logic 1, the MPU can read from memory.

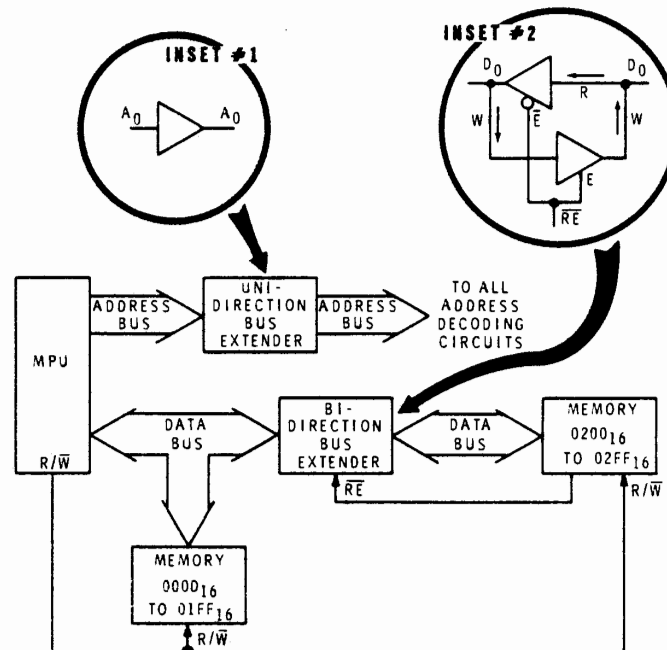
In small microprocessor systems (less than ten devices), the MPU and its support IC's can be connected directly together. However, when more circuits are added to the address and data busses, the MPU can no longer supply the necessary current.

To solve this problem, bus extenders (buffers) are connected between the MPU and most of the surrounding IC's. These supply the necessary drive. Figure 10-12 illustrates a typical circuit.

The address bus extender is uni-directional. That is, it passes a signal in only one direction. It consists of 16 individual buffer drivers; one for each address line.

Unlike address signals, data signals can originate at the MPU or in peripheral circuits such as memory. Therefore, *bi-directional* bus extenders are required. Each data bus extender consists of two 3-state buffer drivers wired back-to-back as shown in Inset 2 of Figure 10-12. The 3-state feature in this case is complementary. That is, when the read enable ( $\overline{RE}$ ) control line is low, the read buffer is enabled, and when the  $\overline{RE}$  line is high, the write buffer is enabled. Remember, the terms read and write are always expressed in relation to the MPU.

Figure 10-12  
Block diagram of typical microproces-  
sor system using bus extenders.





You may wonder why an  $\overline{RE}$  line is required in addition to the  $R/\overline{W}$  line. It wouldn't be, if all the memory and any other bus operated devices were electrically connected **outside** of the bus extenders. However, in most systems, there are some circuits connected directly to the MPU, with additional circuits connected to bus extenders. Thus, the  $\overline{RE}$  control is necessary for valid logic transfer.

During a read cycle, data is transferred to the MPU from memory or other support devices. When a memory location "down line" from the bus extender is addressed, the bus extender will receive an  $\overline{RE}$  logic 0 signal, which enables the buffer in the "read" direction. All devices not addressed, before or after the bus extender, will be 3-stated into their open circuit configuration. In this case, the  $R/\overline{W}$  control could have been inverted and connected to the bus extender  $\overline{RE}$  input. However, if memory "up line" from the bus extender (MPU side) is addressed, the bus extender must remain in its "write" configuration. It could not do this if the inverted  $R/\overline{W}$  control line was used.

Since all of the devices "down line" are 3-stated to their open-circuit condition, the input to the "read" buffer of the bus extender would be undefined and its output would assume a logic level (usually logic 1 for TTL gates) that could interfere with data transfer. By using an  $\overline{RE}$  control signal not totally defined by the  $R/\overline{W}$  control, the bus extender can be forced into its "write" state and prevent any "down-line" interference.

The circuit you constructed from Figure 10-10 used the  $\overline{CE}$  and  $R/\overline{W}$  signals to produce the necessary  $\overline{RE}$  signal. This is then used to control the bus extender where it interfaces the data connector blocks with the MPU data bus in the Trainer. As shown in Figure 10-10, the  $\overline{CE}$  signal from pin 8 of IC1 is inverted by IC5A. Thus, when memory IC's 3 and 4 are enabled and the  $R/\overline{W}$  line is logic 1, the output of IC5B is logic 0, which enables the bus extender "read" buffer. If IC's 3 and 4 are not enabled, the  $\overline{RE}$  line remains high regardless of  $R/\overline{W}$  level, and the bus extender remains in its "write" condition.

The final section of this experiment will examine a method for determining the validity of memory. First however, you will learn to read an assembled program listing. This format will be used from now on in these experiments. What you will see is a photocopy of each program as it is assembled and printed by a computer. This serves two purposes: First, it insures that no typographical errors have been introduced during manual reproduction. Second, it gives you an opportunity to become familiar with the format used by most periodicals and books for program listing.

All of the following programs were assembled with a Motorola EXORciser® and printed with a Digital Equipment Corporation decwriter II®. As shown in Figure 10-13, each listing can be divided into two main sections. The right half contains the assembly language program just as the programmer typed it into the computer. The left half of the listing was produced (assembled) from the data in the right half of the listing. This half contains the machine language code that must be entered into the Microprocessor Trainer.

The program listing contains eight columns of information. A brief explanation of each follows.

- Column 1 is a sequential list of numbers produced only as a reference to identify listing lines.

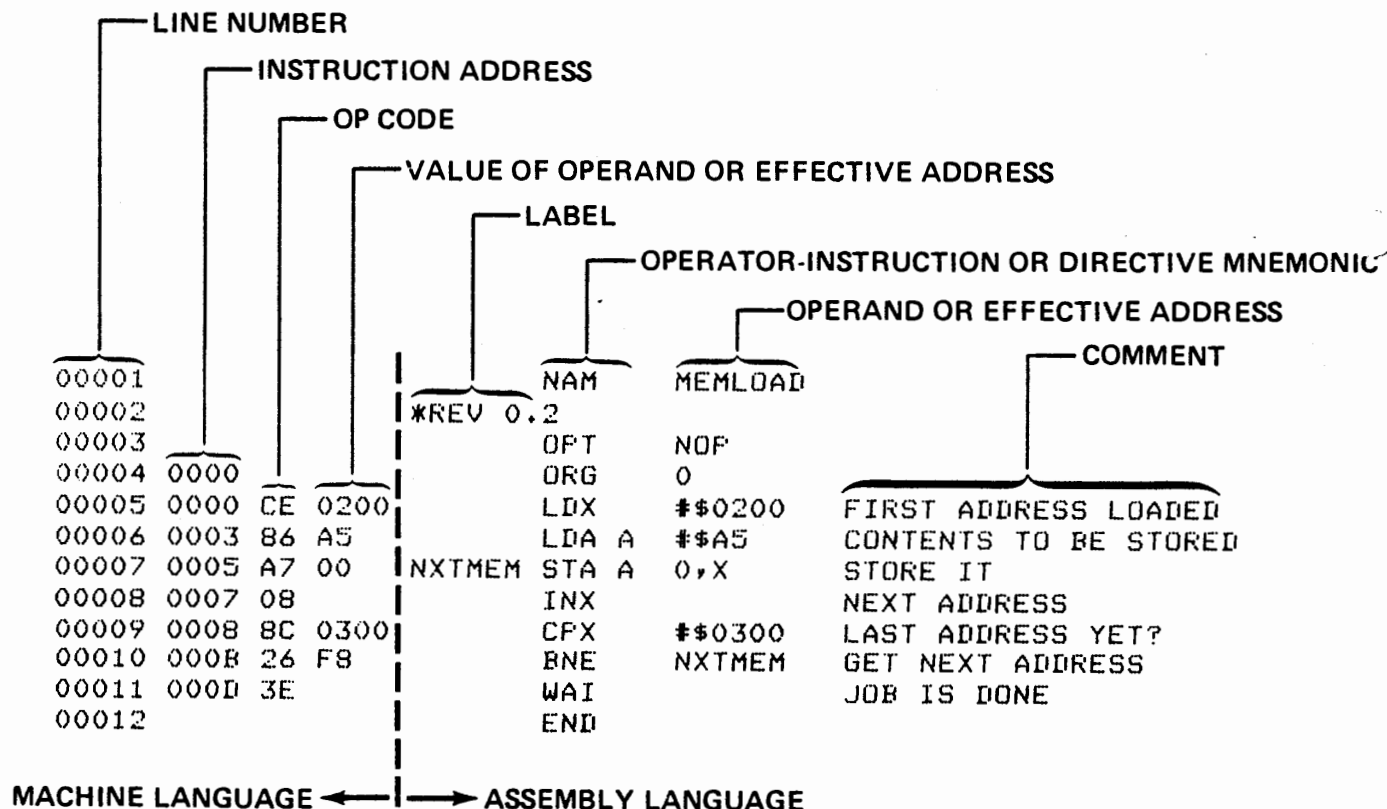


Figure 10-13

Assembled program for loading data  
into a block of memory.

- Column 2 lists each instruction address. Depending on the number of bytes in each instruction, these addresses can be sequential, or spaced 2 or 3 addresses apart. In Figure 10-13, line 00005 contains instruction address 0000, while line 00006 contains instruction address 0003. This occurred because op code CE required two address bytes 02 and 00 to complete the instruction.
- Column 3 lists instruction op codes only. This accounts for the variable address spacing in column 2.
- Column 4 contains the operand or effective address. Thus, this column may have none, one, or two bytes of data, depending on the op code.
- Column 5 begins the assembly language portion of the program listing. It contains any labels used to assemble the program. If the label is preceded by an asterisk (\*), the following information is a comment only, and not used for program assembly. In Figure 10-13, line 00002 contains the label "\*REV 0.2". This is a comment that states this is a program that has been revised two times. It serves only as a handy reference for the programmer to keep track of his program status.

If the label is not preceded by an asterisk, the label becomes a way of finding an address in a branch or jump routine. Line 00010 contains the instruction BNE followed by NXTMEM. This says, branch if not equal to the address defined by the label NXTMEM. Thus, the program will jump to the address at line 00007, where the label NXTMEM is located.

- Column 6 lists the operator-instruction for the program, or the directive mnemonic used by the assembler.
- Column 7 lists the operand or the effective address of the instruction or directive.
- Column 8 contains any comments the programmer wishes to make. These are usually a description of the program steps to aid in understanding the program.

More information on assembly programming is available in Motorola's "M6800 Microprocessor Applications Manual" (Heath # EDP-244). For the purpose of this section, we will be primarily concerned with machine coding, columns 2, 3, and 4.

## Procedure (continued)

28. Refer to Figure 10-13 and enter the program beginning at address 0000<sub>16</sub>.
29. Examine your program beginning at address 0000<sub>16</sub>. The addresses and their contents should agree with the list in Figure 10-14.
30. Press RESET, then press DO and enter address 0000<sub>16</sub>. The display will go dark, indicating the microprocessor is working.
31. Press RESET and examine a number of memory locations between 0200<sub>16</sub> and 02FF<sub>16</sub>. The contents will be A5<sub>16</sub>. You should be able to single-step through each memory location and verify that it functions properly by observing that A5<sub>16</sub> is stored at each address. By changing the contents at address 0004<sub>16</sub> to a different value, say 5A<sub>16</sub>, and executing the program you can verify that none of the memory defaulted to the original value, A5<sub>16</sub>.
32. This completes the experiment. However, DO NOT disconnect the memory circuit from the Trainer. You will use this circuit in Experiment 3. Proceed to Experiment 2.

ADDRESS	DATA
0000	CE
0001	02
0002	00
0003	86
0004	A5
0005	A7
0006	00
0007	08
0008	8C
0009	03
000A	00
000B	26
000C	F8
000D	3E

Figure 10-14  
Data listing for program in Figure  
10-13.

## Experiment 2

### CLOCK

#### OBJECTIVES:

*Show how the interrupt request can be implemented.*

*Show how an external timing signal can synchronize the MPU.*

### Introduction

In this experiment, you will improve the clock program you developed earlier. A line frequency (60 Hz) signal provides timing accuracy to the clock. The line frequency signal is connected to the interrupt request ( $\overline{\text{IRQ}}$ ) input. This makes the clock extremely accurate.

### Materials Required

- 1 ET-3400 Microprocessor Trainer (with hard-wired circuit).
- 1 6" hookup wire.

### Procedure

#### Programming Notes:

- Begin program (listed in Figure 10-15) at address  $0003_{16}$  (line 00009) and enter data  $CE_{16}$ . The first three address locations are reserved for clock display time and will be entered after the main program has been entered.
- As you load the program, notice that no address is specified at lines 00008, 00015, 00023, 00028, 00041, 00047, and 00052. These lines are used for comments or assembler directives. Just follow the program address and ignore these lines.
- Line 00049 contains an ORG (originate) statement which is an assembler directive. Therefore, you can ignore the address specified. The next instruction you must enter goes to memory location  $00F7_{16}$ . Use the EXAM and CHAN keys to enter the opcode.

```

00001          NAM      CLOCK-2 * REV 0.6
00002          **LINE  ACCURACY CLOCK PROGRAM
00003          OPT      NOP
00004 0000          ORG      0
00005 0000 0001      SECOND RMB 1
00006 0001 0001      MINUTE RMB 1
00007 0002 0001      HOURS  RMB 1
00008          ** INTERRUPT HANDLING
00009 0003 CE 003D    TIMPAS LDX  #$003D  61
00010 0006 09        ONE60T DEX          TIME TICKING OFF
00011 0007 27 04      BEQ      TIMEUP    60 PULSES YET?
00012 0009 0E        CLI
00013 000A 3E        WAI          WAITING
00014 000B 20 F9      BRA      ONE60T    GO BACK & WAIT AGAIN!
00015          ** INCR ONE SECOND AND UPDATE
00016 000D C6 60      TIMEUP LDA B  #$60    SIXTY SECONDS,SIXTY MINUTES
00017 000F 0D        SEC          ALWAYS INCREMENT SECONDS
00018 0010 8D 11      BSR      INCR      INCREMENT SECONDS
00019 0012 8D 0F      BSR      INCR      INCREMENT MINUTES IF NEEDED
00020 0014 C6 13      LDA B  #$13    TWELVE HOUR CLOCK
00021 0016 8D 0B      BSR      INCR      INCREMENT HOURS
00022 0018 8D FCBC    JSR      REDIS     RESET DISPLAYS
00023          FCBC    REDIS EQU  $FCBC
00024 001B 8D 17      BSR      PRINT
00025 001D 8D 15      BSR      PRINT
00026 001F 8D 13      BSR      PRINT    PRINT HOURS,MINUTES,SECONDS
00027 0021 20 E0      BRA      TIMPAS    DO IT ALL AGAIN
00028          ** INCR - INCREMENT SUBROUTINE
00029 0023 A6 00      INCR  LDA A  0,X    DATA WORD INTO A
00030 0025 89 00      ADC A  #0          INCREMENT IF NECESSARY
00031 0027 19        DAA          FIX TO DECIMAL
00032 0028 11        CBA          TIME TO CLEAR?
00033 0029 25 01      BCS      INC1      NO
00034 002B 4F        CLR A
00035 002C A7 00      INC1  STA A  0,X
00036 002E 08        INX
00037 002F 07        TPA
00038 0030 88 01      EOR A  #1          COMPLEMENT CARRY BIT
00039 0032 06        TAP
00040 0033 39        RTS
00041          ** PRINT - PRINT HEX BYTES
00042 0034 09        PRINT DEX          POINT X AT BYTE
00043 0035 96 02      LDA A  $02        WHAT'S IN HOURS?
00044 0037 27 05      BEQ      ADJUST    IF IT'S ZERO
00045 0039 A6 00      CONTIN LDA A  0,X
00046 003B 7E FE20    JMP      OUTBYT
00047          FE20    OUTBYT EQU  $FE20    MONITOR ROUTINE
00048 003E 7C 0002    ADJUST INC  HOURS    MAKE IT ONE
00049 0041 20 F6      BRA      CONTIN    RESUME
00050 00F7          ORG      $00F7
00051 00F7 3B        RTI
00052          END

```

Figure 10-15

Assembled program for real-time  
clock.

## Procedure (continued)

1. If you have not done so, switch the Trainer on. Then enter the program listed in Figure 10-15.
2. Refer to Figure 10-16 and install the 6" hookup wire between the LINE socket and the  $\overline{\text{IRQ}}$  socket. Do not disturb the other circuit you have wired into the Trainer.
3. Your clock is ready to run. Determine at what time you wish to start the clock; then enter the seconds at address  $0000_{16}$ , the minutes at address  $0001_{16}$ , and the hours at address  $0002_{16}$ . For example, to set the clock for 9:25:30, enter the following:

Address	Data
0000	30 (seconds)
0001	25 (minutes)
0002	09 (hours)

4. Press RESET, then press DO and then enter the first 3 digits of the starting address ( $000_{16}$ ). As the time you have set approaches, enter the fourth digit (3), but do not release the key. At precisely the correct time, release the "3" key. The display will momentarily go dark, and then show the correct time, with the seconds digit updating at a 1-second rate.

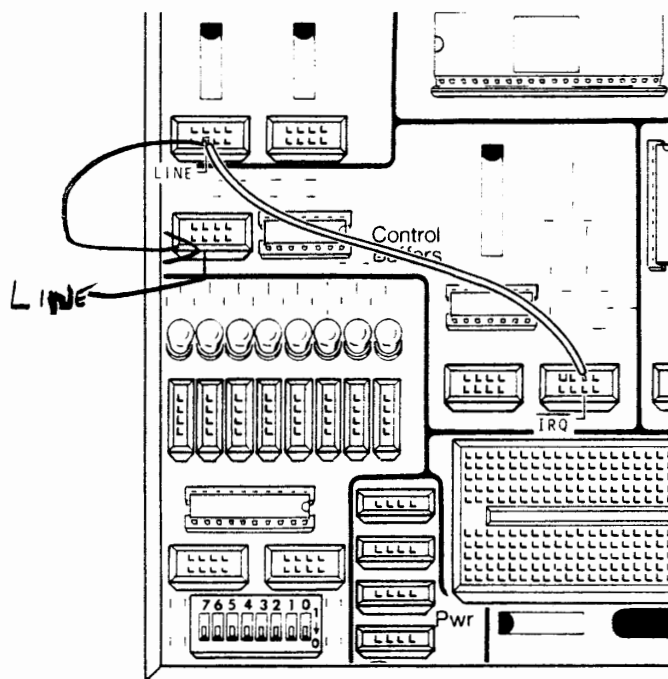


Figure 10-16  
Clock interrupt request line connection.

## Discussion

Although it appears to be a simple process to connect the LINE signal to the  $\overline{\text{IRQ}}$  input of the microprocessor, AC line voltage had to be reduced in amplitude and processed into a waveform acceptable to the microprocessor. The circuit used by the Trainer is shown in Figure 10-17A. It uses a comparator with positive feedback to process the AC signal.

A sample of AC line signal is coupled through current limiting resistors R1 and R2 to the negative input of the comparator. Diode D1 limits the negative swing of the AC signal to approximately 0.7 volts. The comparator tracks the AC input and switches logic levels at its output, at the same rate. Positive feedback through resistor R6 speeds up the rise and fall times at the output.

A simpler circuit is shown in Figure 10-17B. Input current is limited by resistor R1, while diodes D1 and D2 limit the voltage swing within TTL levels. As before, the line frequency can be obtained from the secondary of a power transformer. The NAND gate is used to buffer the input signal and provide some speed-up of the rise and fall times.

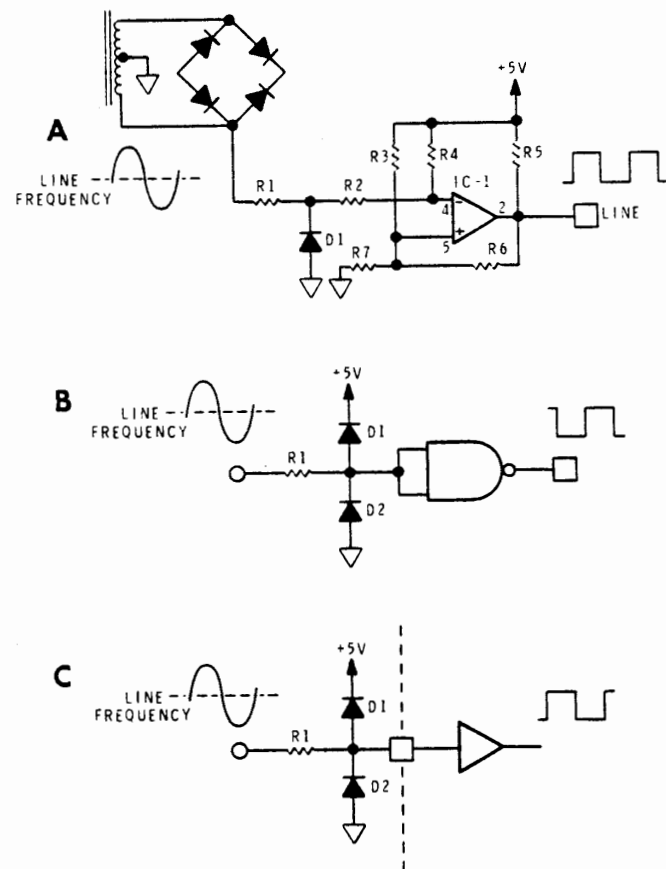


Figure 10-17  
Line frequency signal processing.



If a buffer is already supplied in the system, as in Figure 10-17C, then it is only necessary to limit the signal current and voltage swing with a resistor and two diodes.

The clock program (Figure 10-15) is broken down into five main sections. These are:

Lines 00005-00007. Reserved addresses where the starting seconds, minutes, and hours are stored. As the program progresses, these addresses are updated to current time.

Lines 00009-00014, and 00051. They handle the interrupt routine, and count the seconds.

Lines 00016-00027. The main part of the program keeps track of seconds; increments seconds, minutes, and hours when appropriate.

Lines 00029-00040. A subroutine that handles the mathematical and updating part of the program.

Lines 00042-00049. Another subroutine that updates the display with new data. Also insures that hours never go to zero.

In addition, a number of monitor subroutines are used.

Since this experiment is concerned primarily with interrupt handling, this discussion will explain only that part of the program in detail.

Line 00009 — Load the index register with the line frequency plus one. (Line frequency is  $60_{10}$ , plus 1 yields  $61_{10}$  or  $3D_{16}$ .) Thus, when the index register is decremented in the next instruction, the count circuit will provide a precise division by 60.

Line 00010 — Decrement the index register by one. Clock timing has begun.

Line 00011 — The first time through, the index is not zero. Therefore, the branch instruction is not executed. On a later pass, when the index register is zero, the program will branch to TIMEUP.

Line 00013 — Wait for the interrupt request to arrive.

As discussed in Unit 6, when a non maskable interrupt ( $\overline{\text{NMI}}$ ) occurs, the contents of the index register, program counter, accumulators, and condition code register are stored in the stack. The program counter is then loaded with a new address that is found at addresses  $\text{FFFFC}_{16}$  and  $\text{FFFD}_{16}$  (located in ROM). If you examine these addresses you will find  $00_{16}$  and  $\text{FD}_{16}$  respectively. The microprocessor will then execute the instruction at address  $00\text{FD}_{16}$ .

Line 00050 — Return from interrupt. When this instruction is executed, the microprocessor retrieves the data previously stored in the stack. This includes the index register and program counter contents. It then executes the instruction pointed to by the program counter.

Line 00013 — Branch always is the instruction immediately following the wait for interrupt. This sends the microprocessor back to line 00010 ( $\text{ONE60T}$ ).

At this point, you should notice that the program is in a loop that repeats every sixtieth of a second. This will continue until the index register decrements to zero. When zero is attained, the branch-if-zero instruction will send the microprocessor off to the main part of the program, which increments the clock by one second. The interrupt routine repeats again after the clock advances.

The remainder of the program is very similar to the clock program presented earlier. A full explanation of the techniques used was discussed in a previous experiment and is not repeated here.

This completes this experiment. Switch the Trainer power off and remove the wire between LINE and  $\overline{\text{IRQ}}$ . Do not disturb the remaining wires. The circuit you previously constructed will be used in Experiment 3. Proceed with Experiment 3.

## Experiment 3

### ADDRESS DECODING

#### OBJECTIVES:

*Demonstrate the difference between full and partial address decoding.*

*Show how an address decoding chart is assembled.*

*Demonstrate how an address can be decoded using various types of logic circuits.*

*Show how to construct a memory address map.*

#### Introduction

Many different combinational logic circuits can be used to decode binary bit patterns. We will examine several decoding techniques in this experiment. The first example will use the circuit you wired in the first experiment.

#### Material Required

- 1 Microprocessor Trainer (with hard-wired circuit)
- 1 1000 ohm, 1/4-watt, 10% resistor
- 1 6" double-sided foam tape.
- 1 Large connector block
- 1 74LS42 integrated circuit (443-807)
- 1 74LS266 integrated circuit (443-719)

Hookup wire

## Procedure

1. Carefully examine the circuit you wired to the Trainer in Experiment 1 to see if any wires have pulled out. Figure 10-18 is an electrical diagram of the circuit.
2. Load the program listed in Figure 10-19 beginning at address  $0000_{16}$  with data  $CE_{16}$  and ending at address  $0019_{16}$  with data  $E6_{16}$ .
3. Press RESET, then press DO and enter  $0000_{16}$ . The display will go out. After a short period of time, all display segments and decimal points will light. This is an indication that the program has been executed.

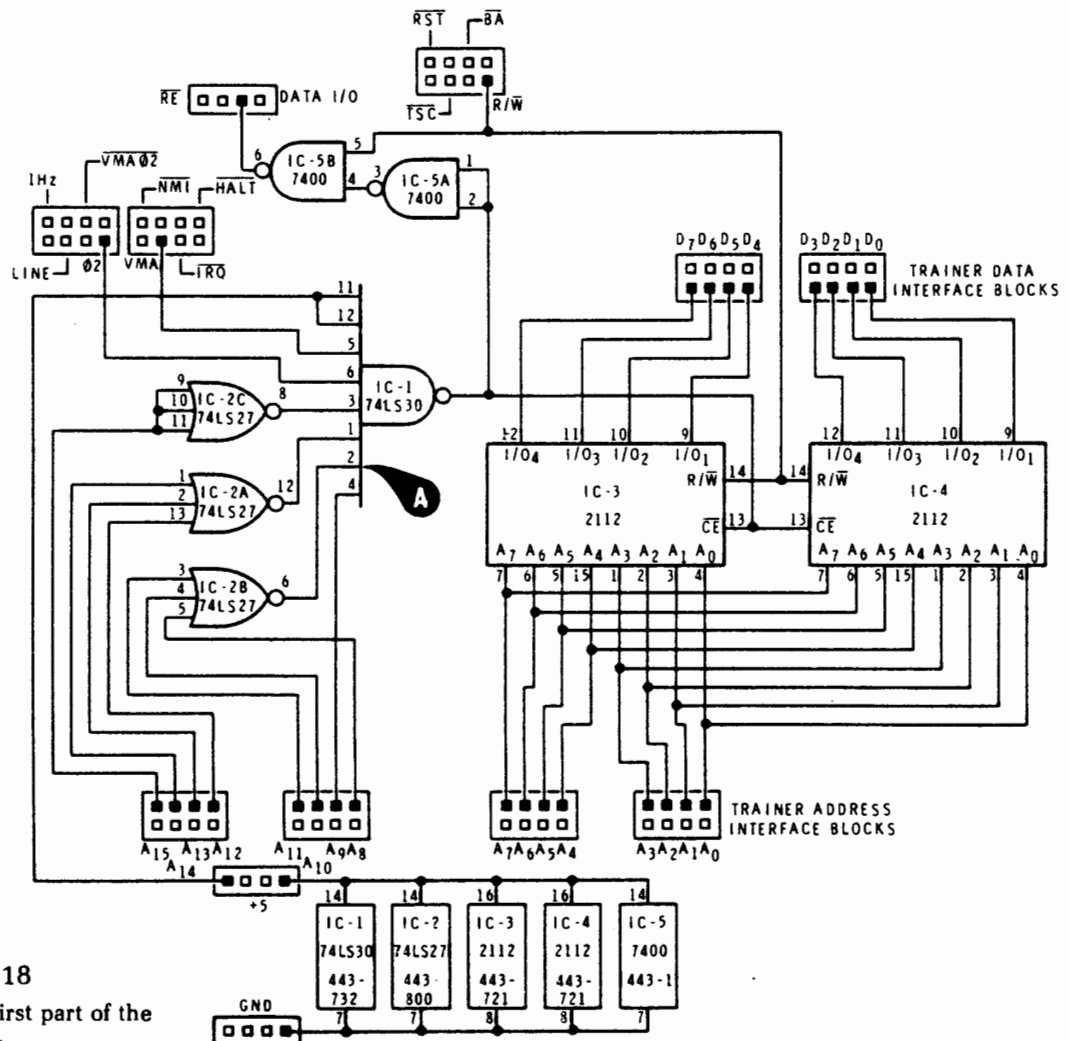


Figure 10-18

**Circuit diagram of the first part of the decoding experiment.**

```

00001      NAM      DECODECK REV. 0.3
00002      OPT      NOP
00003 0000 CE 001A REDO   LDX      $$001A    1ST BLOCK, 1ST ADR
00004 0003 86 BB        LDA A    $$BB      DATA TO BE STORED
00005 0005 A7 00   LOAD1 STA A    X        STORE IT
00006 0007 08        INX          POINT TO NEXT ADR
00007 0008 8C 0200     CPX      $$0200    1ST BLOCK, LAST ADR(+1)
00008 000B 26 F8      BNE      LOAD1
00009 000D CE 0300     LDX      $$0300    2ND BLOCK, FIRST ADR
00010 0010 A7 00   LOAD2 STA A    X        STORE IT
00011 0012 08        INX          POINT TO NEXT ADR
00012 0013 8C 0000     CPX      $0000    2ND BLOCK, LAST ADR(+1)
00013 0016 26 F8      BNE      LOAD2
00014 0018 20 E6      BRA      REDO      RECYCLE
00015      END

```

Figure 10-19  
Program for memory decoding experiment.

4. You have written  $BB_{16}$  into every memory location except where the program resides, and between address  $0200_{16}$  thru  $02FF_{16}$ . Verify this by examining a number of locations between  $001A_{16}$  and  $01FF_{16}$ . Now examine a number of locations between  $0200_{16}$  and  $02FF_{16}$  (hard-wired RAM.)

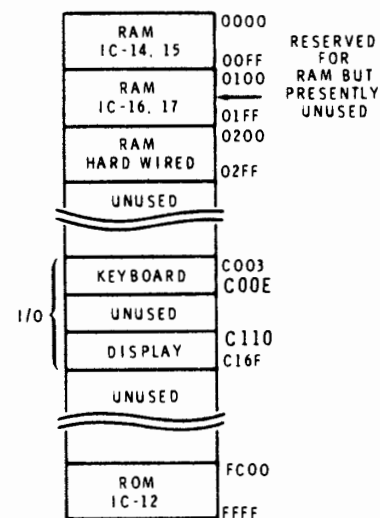
NOTE: Addresses  $00D3_{16}$  thru  $00F3_{16}$  will not contain  $BB_{16}$ . The Trainer monitor routine uses that portion of RAM.

## Discussion

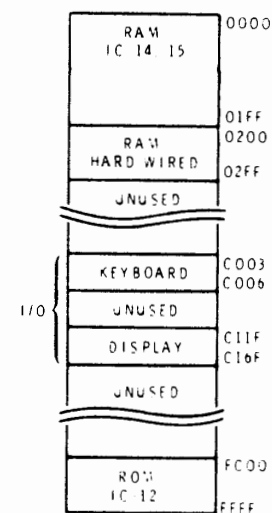
The memory you have hard-wired to the Trainer occupies memory space directly above that allocated for "on-board" memory. The assignments are shown in Figure 10-20. In Experiment 1, you manipulated data within this range. Then you wrote a program to load these addresses ( $0200_{16}$  thru  $02FF_{16}$ ) with data. Thus, you found that each space in memory responds to a specific address.

In this experiment, you tried to load data into every possible memory location except the program location and memory block  $0200_{16}$  thru  $02FF_{16}$ . You were unsuccessful with the addresses that do not presently contain RAM, if your Trainer is an ET-3400. Again, refer to Figure 10-20.

When addresses  $0200_{16}$  thru  $02FF_{16}$  were checked, the contents were not modified by the program. This proves that this section of memory is fully decoded. That is, each location can be accessed with only one specific address.



A. ET-3400 MEMORY MAP



B. ET-3400A MEMORY MAP

Figure 10-20

Memory maps of the Microprocessor Trainers with additional off-board RAM at  $0200_{16}$  through  $02FF_{16}$ .

Although it was described in the Trainer assembly manual, now is a good time to briefly look at the Trainer display. As shown in Figure 10-20, the display occupies space in the Trainer memory network. In addition, each display segment and decimal point responds to a specific address. Thus, when you entered  $BB_{16}$  between  $0300_{16}$  and  $FFFF_{16}$  in memory, you also wrote into each display data latch. This is why all of the display segments lit while the program was running.

A decoding chart such as the one shown in Figure 10-21 can be used to indicate the address code for a memory location. The 1's and 0's in the high byte indicate the logic levels required to enable the memory block, while the X's in the low byte indicate that either a 1 or 0 may be present to select the actual address. Notice that all 16 bits help determine a specific address, which indicates this memory is fully decoded.

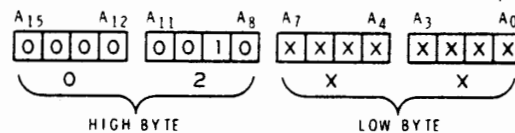


Figure 10-21

Decoding chart for a memory circuit that is fully decoding. Includes memory from  $0200_{16}$  to  $02FF_{16}$ .

## Procedure (continued)

5. Pull the wire end at location A (see Figure 10-18) from the connector block. Location A is pin 2 of IC1.
6. Now press DO and reexecute the program beginning at address  $0000_{16}$ . Again the display will go out, and then light all segments and decimal points after a short period of time.
7. Press RESET and examine a number of addresses between  $0200_{16}$  and  $02FF_{16}$ . Each address should now contain data  $BB_{16}$ .
8. Change the data at address  $0210_{16}$  to  $AA_{16}$ .

9. Examine and record the data stored at the following addresses.

0010 --	0810 --
0110 --	0910 --
0210 --	0A10 --
0310 --	0B10 --
0410 --	0C10 --
0510 --	0D10 --
0610 --	0E10 --
0710 --	0F10 --

## Discussion

When you disconnected IC2B from the circuit, address lines  $A_8$ ,  $A_{10}$ , and  $A_{11}$  were no longer able to take part in address decoding. Since line  $A_9$  was still connected, the circuit still decodes to  $02XX_{16}$ . However, other addresses will now decode the same circuit.

A new decoding chart (see Figure 10-22) can be assembled by examining the schematic in Figure 10-18. Bits  $A_0$  thru  $A_7$  are connected directly to memory and select specific addresses in that memory block. An X is placed in each of the corresponding boxes to indicate that the logic levels are unknown but do take part in address selection. Bits  $A_{12}$  thru  $A_{15}$  must be logic 0 so the correct logic level will be applied to the inputs of NAND gate IC1. Therefore, a 0 is placed in each box.

Disconnecting IC2B removed bits  $A_8$ ,  $A_{10}$ , and  $A_{11}$  from the circuit. Since these bits have no effect in the decoding process, these are "don't care" bits. A dot (•) symbol is then placed in each of the corresponding boxes.

Address bit  $A_9$  is still connected to IC1. Since it must be a logic 1 to enable memory, a 1 is placed in its box.

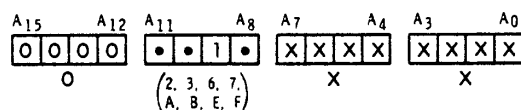
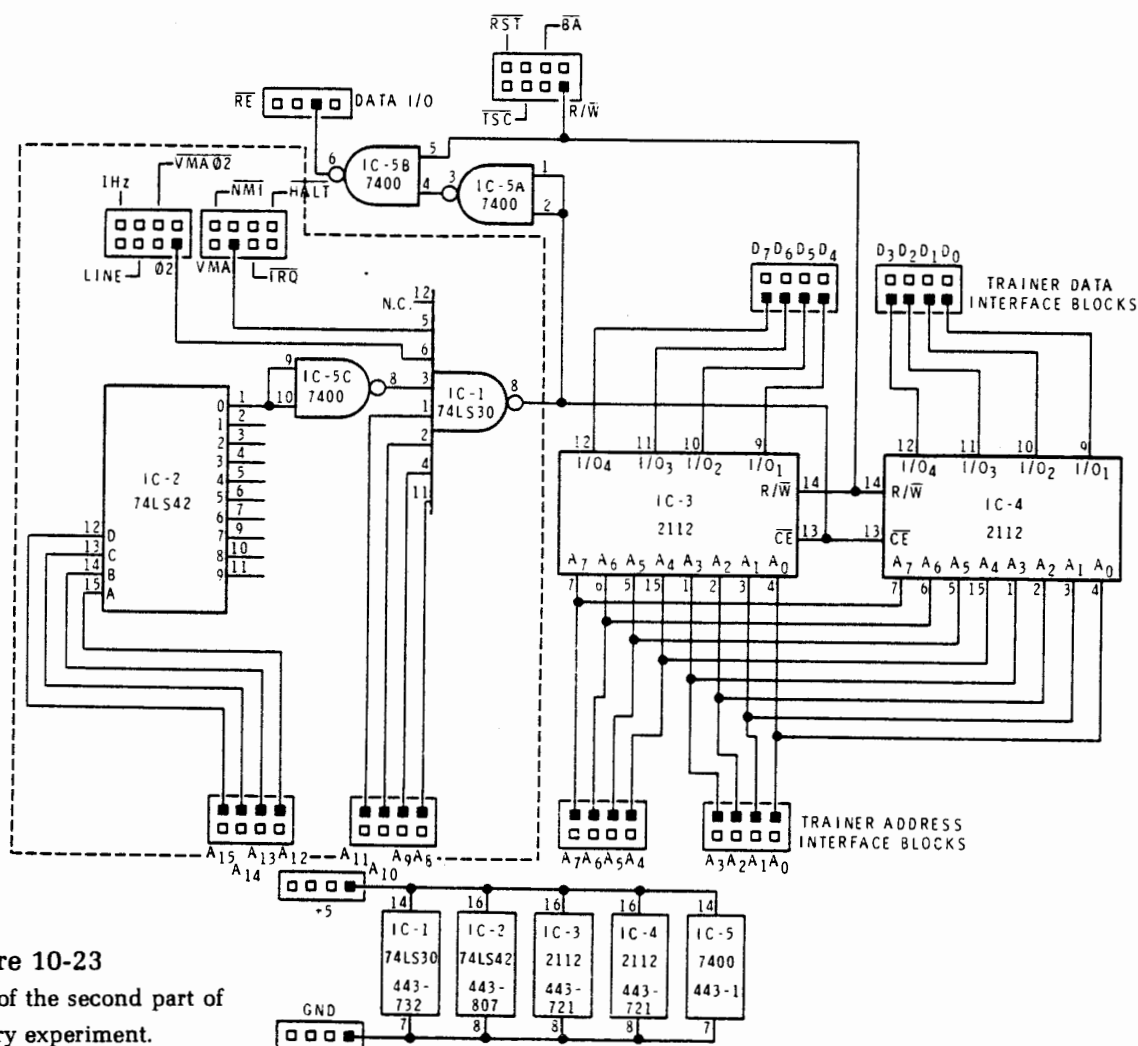


Figure 10-22

Decoding chart for a memory circuit that is partially decoded. Indicates that many addresses will decode this block of memory.

Once you have constructed a decoding chart, you can determine all of the addresses that will access the circuit. In this case, the most significant four bits are clearly defined as zeroes. The next four bits are more complicated. The only defined bit is  $A_9$ . Bits  $A_8$ ,  $A_{10}$ , and  $A_{11}$  can be any logic level. Therefore, any hex number which contains the  $A_9$  bit as a logic 1 will be valid. These hex numbers include 2, 3, 6, 7, A, B, E, and F. Bits  $A_0$  through  $A_7$  are variable and select the specific addresses within the circuit.

The chart you completed in step 9 will support this discussion. When you changed the contents of address  $0210_{16}$  to  $AA_{16}$ , the addresses that had 3, 6, 7, A, B, E, and F as the second most significant hex digit also appeared to contain  $AA_{16}$ . This, of course is impossible, since no memory exists for those addresses.



**Figure 10-23**  
Circuit diagram of the second part of  
the memory experiment.



## Procedure (continued)

10. Switch the Trainer power off. Then carefully remove all of the wires from IC1 and IC2, except for the +5 V and ground wires. The remaining circuit wires will remain unchanged in the circuit you are about to construct. To aid you, the new circuit is enclosed by a dashed line in Figure 10-23.
11. Using the IC puller tool, carefully remove IC2 (74LS27). Then install the 74LS42 16-pin IC. Since you are replacing a 14-pin IC with a 16-pin IC, you will have to reposition the +5V wires or ground wires (pin 8 or pin 16).
12. Refer to Figure 10-23 and wire IC1 and IC2 into the circuit. Note that gate C of IC5 is also wired into the circuit.

DEC BCD INPUT					OUTPUT LINES									
NO.	D	C	B	A	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	1	1	1	1	1	1	1	1	1
1	0	0	0	1	1	0	1	1	1	1	1	1	1	1
2	0	0	1	0	1	1	0	1	1	1	1	1	1	1
3	0	0	1	1	1	1	1	0	1	1	1	1	1	1
4	0	1	0	0	1	1	1	1	0	1	1	1	1	1
5	0	1	0	1	1	1	1	1	1	0	1	1	1	1
6	0	1	1	0	1	1	1	1	1	1	0	1	1	1
7	0	1	1	1	1	1	1	1	1	1	1	0	1	1
8	1	0	0	0	1	1	1	1	1	1	1	1	0	1
9	1	0	0	1	1	1	1	1	1	1	1	1	1	0
>9	INVALID CODES				1	1	1	1	1	1	1	1	1	1

Figure 10-24

Logic truth table for 74LS42 4-to-10  
line decoder.

13. Carefully examine the circuit to make sure all of the wires are properly routed. Also make sure none of the previously installed wires have pulled free.
14. Using the schematic in Figure 10-23, and the logic truth table for IC2, found in Figure 10-24, construct a memory decoding chart in Figure 10-25.

A <sub>15</sub>	A <sub>12</sub>	A <sub>11</sub>	A <sub>8</sub>	A <sub>7</sub>	A <sub>4</sub>	A <sub>3</sub>	A <sub>0</sub>
<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Figure 10-25

Blank decoding chart.

15. Is the circuit fully or partially decoded?  
 \_\_\_ Fully \_\_\_ Partially  
 Why? \_\_\_\_\_  
 \_\_\_\_\_
16. Now that you have determined the address block your memory resides at, load a few of the addresses with data.

## Discussion

The circuit you just constructed contains a fully decoded memory. Its decoding chart is shown in Figure 10-26. Each bit position is used to define a specific memory address.

In an earlier experiment, you used a combinational logic decoding technique. With that technique, it is necessary to use an individual logic circuit for each memory block. In the circuit you just completed, a 4-to-10 line decoding IC was used to define a block of memory. Since it is possible to define ten different values with four input variables, this device could be used to address ten different memory blocks. Although there was no difference in the number of IC's used in the two experiments, expanding the amount of memory would require fewer device select IC's when the 4-to-10 decoder is used. This can be seen by reexamining the circuits in Figures 10-18 and 10-23.

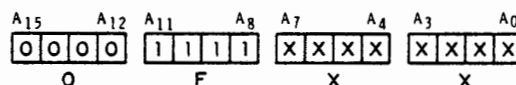


Figure 10-26

Decoding chart for circuit using  
4-to-10 decoding IC.

## Procedure (continued)

17. Switch the Trainer power off. Then disconnect the wire at pin 1 of IC2 (74LS42) and connect it to pin 6 of IC2.
18. The circuit is now fully decoded to address 5FXX<sub>16</sub>. This is because every address bit plays a part in determining a specific memory location. Switch the Trainer power on and examine a number of memory locations between 5F00<sub>16</sub> and 5FFF<sub>16</sub>. Notice that you can store and read data at these locations. However, you can no longer store and read data at 0F00<sub>16</sub> thru 0FFF<sub>16</sub>.

## Discussion

The 4-to-10 line decoder can also be used to quickly change decoding addresses, as you have just done. Keep in mind that this experiment uses only one 4-to-10 line decoder to define the most significant hex number. It is not unusual to see them used at lower order addresses. Your Micro-processor Trainer uses this decoding technique for its "on-board" memory.

## Procedure (continued)

19. At this time, you will need more breadboarding space. Locate the box containing the large connector block. Then refer to part A of Figure 10-27 and install the connector strips supplied with the block. You may have some strips left over.
20. Refer to Figure 10-27B. Then remove the paper backing from the vinyl strip supplied with the block, line up the long edges of the strip and block, and press the sticky side of the vinyl strip against the block.

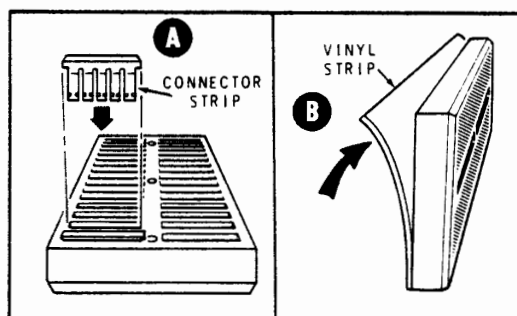


Figure 10-27

Large connector block assembly.

21. Read this whole step, then locate the foam tape and cut two  $\frac{3}{4}$ " squares from it. Refer to Figure 10-28. Remove the paper backing from one side of each square and affix each square to the small edge of the Trainer cabinet as shown. The spacing between the squares should be a little less than the length of the connector block. Now remove the paper backing from the other side of the squares and affix the connector block to the squares. Try to center the squares on the back of the block. This block will provide additional bread-boarding space.
22. Switch the Trainer power off and install a 74LS266 (443-719) IC in the new large connector block, near the left end. This IC will become IC6 in the circuit you construct next.

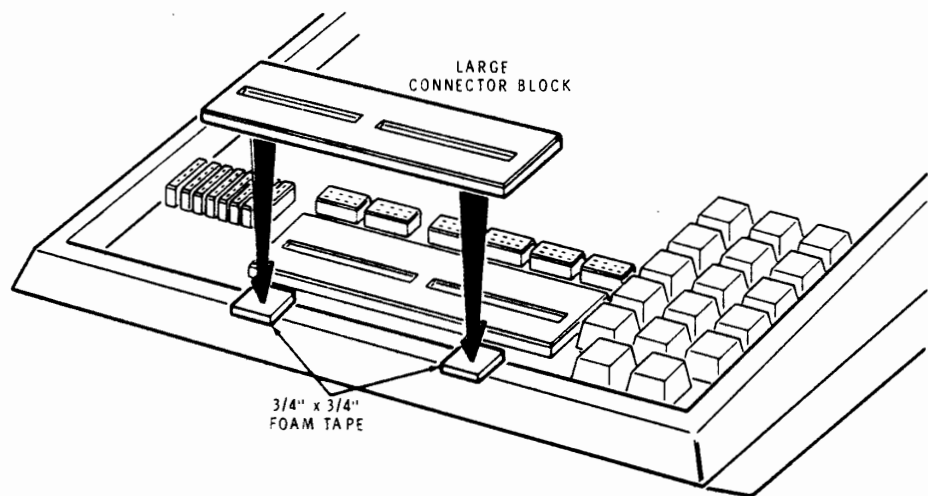
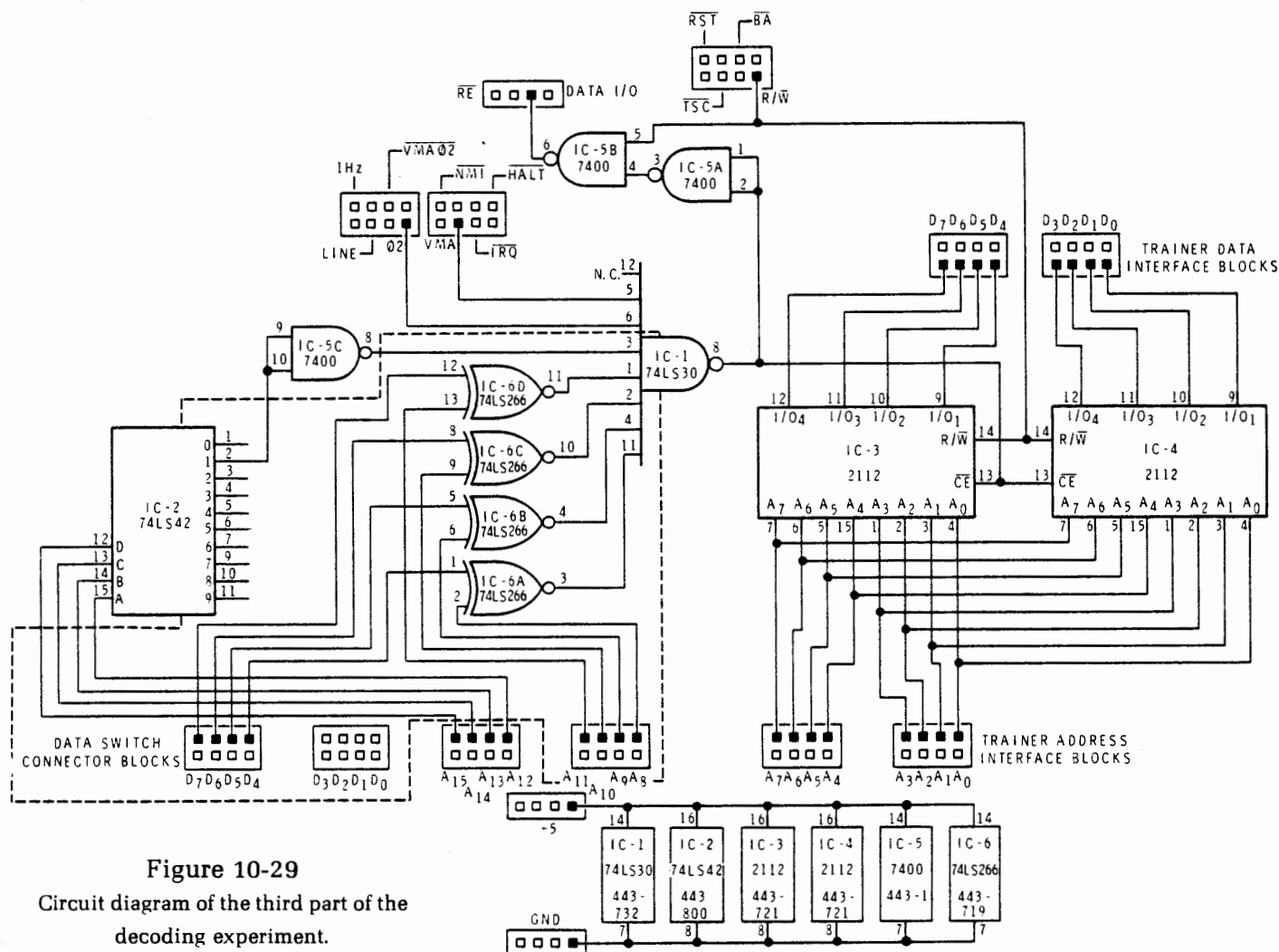


Figure 10-28  
Outboard connector block installation.



**Figure 10-29**

Circuit diagram of the third part of the decoding experiment.

23. Connect +5-volt power to pin 14 and ground to pin 7. Then refer to Figure 10-29 and construct the circuit shown. To aid you, the area enclosed by the dashed line is the only area where modifications are made to the previous circuit.
24. Recheck the wiring to insure it is correct. Be sure pins 9 and 10 of IC5C are connected to pin 2 of IC2.
25. Refer to Figure 10-30. This is the truth table for a gate in IC6. Notice that if the B input is held at logic 0, the relationship between input A and output Y is complementary (A is the inverse of Y). If the B input is held at a logic 1, the relationship between A and Y is direct (no inversion). Thus, input B can be used as a direct driver or an inverter driver. This feature will be used in the experiment.

A	B	Y
0	0	1
1	0	0
0	1	0
1	1	1

$$A \oplus B = Y$$



**Figure 10-30**  
Truth table for exclusive NOR (ENOR) gate.

26. Position all of the data switches in the down (logic 0) position.
27. Using the schematic in Figure 10-29 and the table in Figure 10-30, determine the decoding chart for the circuit you constructed. Fill in the blank decoding chart in Figure 10-31.
28. Is the address fully or partially decoded?  
Fully \_\_\_\_ Partially \_\_\_\_
29. What is the address block this circuit will decode?  
\_\_\_\_<sub>16</sub> thru \_\_\_\_<sub>16</sub>
30. Switch the Trainer on. Then enter data into a number of addresses in the address block you calculated.

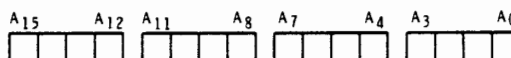


Figure 10-31  
Blank decoding chart.

## Discussion

The exclusive NOR gates used in the circuit all function as inverter drivers. Since each is connected to an individual input in IC1, and the 4-to-10 line decoder is still in the circuit, each address line plays a part in determining a specific address. Therefore, this circuit is fully decoded, and occupies addresses  $1000_{16}$  thru  $10FF_{16}$ .

## Procedure (continued)

31. Change data switches  $D_7$  through  $D_4$  to  $0101_2$ . This equals hex 5.
32. What is the address block this circuit will now decode? \_\_\_\_<sub>16</sub> thru \_\_\_\_<sub>16</sub>
33. Test the new address to see if the circuit will respond properly.

## Discussion

Now you see the power of the exclusive NOR gate. Addresses are easily switched through them. Notice that whatever binary bit pattern you select with the data switch, the circuit responds to it. Now you will see one other feature of these particular exclusive NOR gates.

## Procedure (continued)

34. Switch the Trainer power off. Refer to Figure 10-29. Remove the three wires interconnecting IC1 and IC6, from pin 11 to pin 3, pin 4 to pin 4, and pin 2 to pin 10. Refer to Figure 10-32. Then interconnect pins 3, 4, 10, and 11 of IC6. Finally, insert a 1000 ohm, 1/4-watt, 10% resistor between +5 volts and IC6, pin 11.
35. The circuit should still be located at address block 1500<sub>16</sub> thru 15FF<sub>16</sub>. Check a few of the addresses where you previously entered data. They should contain the same data.
36. Change the data switches to a new bit pattern, determine the address code, then examine a few locations to assure yourself of address code accuracy. Repeat this procedure a number of times.

## Discussion

The 74LS266 exclusive NOR gate has a special characteristic; it has an open-collector output. This means that a number of gates can be tied together, as you just did. The electrical term for this procedure is wire-OR'ing, where the outputs function as though they were inputs to an OR gate.

One more aspect of circuit decoding will be discussed before the next experiment. This is important, since an experimental error could possibly result in the destruction of a gate or memory package.

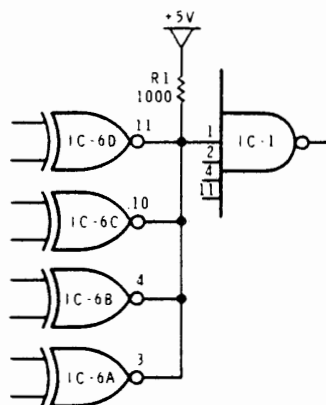


Figure 10-32

Modification to third circuit of experiment.

A problem arises when two circuits decode to the same address. If two memories containing different data occupy the same address, they will try to pull the data lines in two different directions. Since this is electrically impossible, one circuit will give in, usually resulting in permanent destruction to the circuit. Thus, it is important when designing circuits like those used in this experiment to always check for conflicts in address decoding.

### **Procedure (continued)**

37. Switch the Trainer power off and pull the line cord plug.
38. Pull the hookup wires from the circuit and save them for future use.

NOTE: If your Trainer is an ET-3400, perform step 39. If your trainer is an ET-3400A, perform step 39A.

39. Carefully remove IC's 3 and 4 (2112) from the large connector block and install them in IC sockets 16 and 17 in the Trainer. Observe the normal precautions for MOS devices, and make sure you align pin 1 of each IC to the proper position. Then remove all of the remaining components from the two large connector blocks.
- 39A. Carefully remove IC's 3 and 4 (2112) from the large connector block. Observing the precautions for MOS devices, place them on the anti-static pad prior to placing them in the small parts container furnished with this course. Then remove all of the remaining components from the two large connector blocks.

### **Discussion**

Your Microprocessor Trainer now contains  $512_{10}$  bytes of RAM. This is located at addresses  $0000_{16}$  through  $01FF_{16}$ . Proceed to Experiment 4.



## Experiment 4

### DATA OUTPUT

#### OBJECTIVES:

*Demonstrate microprocessor interfacing to an external data display.*

*Show how a 7-segment display is connected.*

*Demonstrate the trade-offs between hardware and software display decoding.*

*Provide an opportunity to write a number of output programs.*

#### Introduction

Until now, you have been using programs that moved data within the Trainer, with any results displayed by the "on-board" LED's. This may be adequate for your purposes, but other methods are needed if external equipment uses the data. The data may take the form of a visual display for an operator to read, or a digital control signal to manipulate an electro-mechanical device. This experiment will present a number of interfacing methods and examine some of the advantages and disadvantages of each method.

#### Material Required

- 1 ET-3400 Microprocessor Trainer
- 8 470 ohm, 1/4-watt, 5% resistors
- 2 10 k ohm, 1/2-watt, 5% resistors
- 1 FND-500 7-segment LED (411-819)
- 1 TIL-312 7-segment LED (411-831)
- 1 7400 integrated circuit (443-1)
- 2 7475 integrated circuits (443-13)

- 1 9368 integrated circuit (443-694)
- 1 74LS30 integrated circuit (443-732)
- 1 74LS27 integrated circuit (443-800)
- 1 74LS259 integrated circuit (443-804)

Hookup wire

## Procedure

- In this part of the experiment, you will examine how the MPU can be interfaced to LED's. Make sure the Trainer power is switched off; then construct the circuit shown in Figure 10-33. Notice that +5 volts and ground are connected to pins 5 and 12 respectively for IC's 1 and 2 (7475). The other IC's use pin 14 for +5 volts and pin 7 for ground.

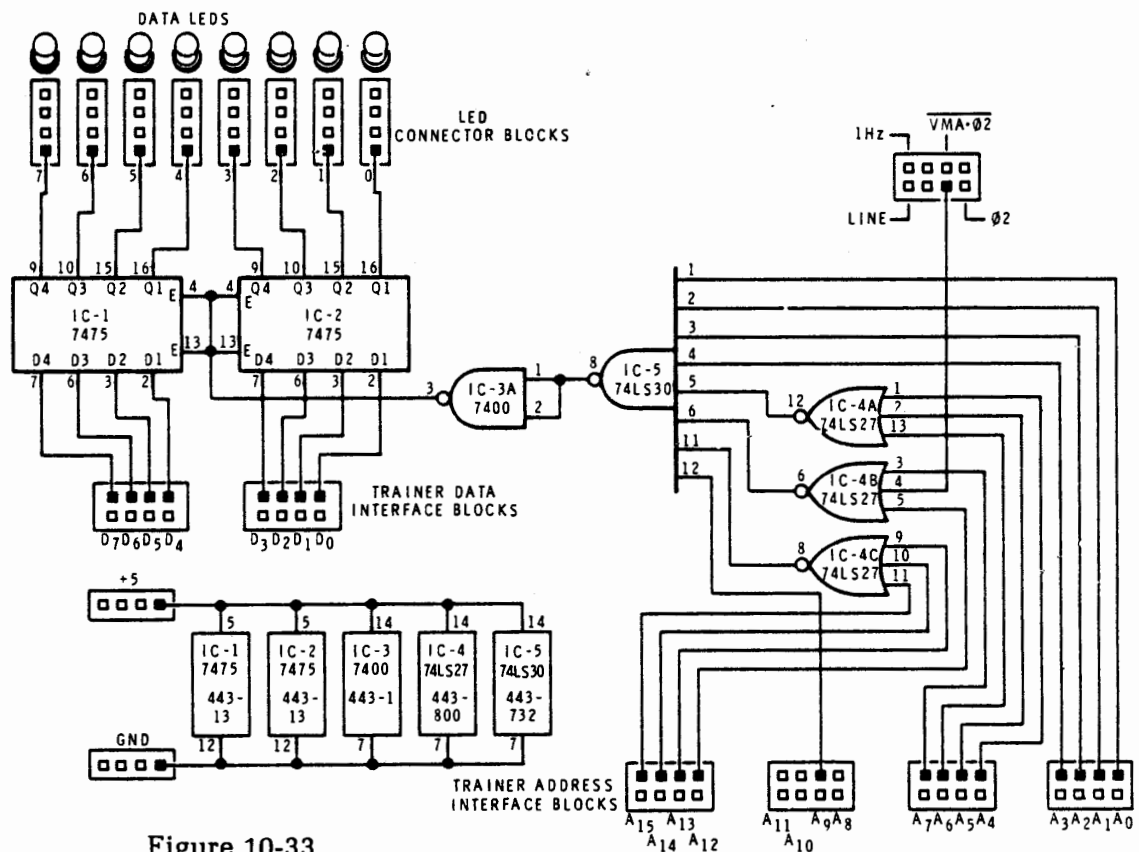
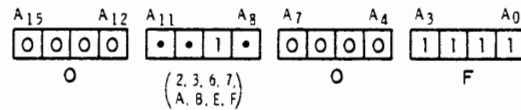


Figure 10-33

Circuit diagram of the first part of the output experiment.

2. Recheck your wiring; then switch the Trainer power on. The data LED's will show a random value.
3. Figure 10-34 is a decoding chart for the circuit you constructed. This shows that the circuit is partially decoded. A 2-digit hex number can be stored at any of these decoded addresses. Examine address  $020F_{16}$ . Then change the contents to  $55_{16}$ .



**Figure 10-34**

Decoding chart for outputting information to the data LED's.

4. The data LED's indicate -----<sub>2</sub>. This is the binary equivalent of the data you stored at address  $020F_{16}$ .
5. What hex value would be required to turn off all of the data LED's? --<sub>16</sub>. Verify your answer.
6. What hex value would be required to turn on all of the data LED's? --<sub>16</sub>. Verify your answer.
7. Change the data at address  $020F_{16}$  a number of times and verify its value with the data LED's.
8. Write and execute a program that will alternately turn all of the data LED's on and off. Use a delay loop in the program so that the on and off cycles can be recognized. Remember that an MPU cycle takes approximately 2.5 microseconds in the unmodified Trainer. If your Trainer has been modified for use with the Memory I/O Accessory, an MPU cycle will take about one microsecond.

If you have any difficulty, use the Trainer single-step function to examine the operation of your program.

## Discussion

Refer to Figures 10-33 and 10-34. Notice that a partial decoding scheme is used. A fully decoded circuit could have been used by adding more combinational logic.

In previous decoding circuits, the VMA and  $\phi 2$  signals were separate. A logic 1 indicated their true state. In this experiment, we took advantage of another Trainer output; the  $\overline{\text{VMA}} \cdot \phi 2$  line. It is logic 0 when both the VMA and  $\phi 2$  signals are at their true state. This reduces the number of logic gates needed for decoding.

The circuit you constructed appears as a **write only memory** to the microprocessor. That is, the MPU can write into the selected address, but it can not read the data stored. However, since eight data LED's monitor the stored information, you can **read** the data. Thus, the MPU is interfaced in a way that produces usable data.

Two bistable quad latch IC's are enabled when one of the eight pre-selected addresses is accessed. They act as an 8-bit memory storage device. Thus, any data appearing on the data lines is latched into the two devices. Since the output of each latch is active, the data LED connected to each will follow the data level. Storing  $00_{16}$  will turn off all of the LED's, while storing  $FF_{16}$  will turn each LED on.

Right now, the data LED's should be switching on and off at a regular interval, because of the program you wrote and executed. If you had any difficulty with the program, refer to Figure 10-35. It lists a program to flash the data LED's. The contents of addresses 0005 and 0006 control the amount of delay. You may wish to change this number if your Trainer has been modified. While this program may not match your program, it is one of many ways to accomplish the same objective.

00001		NAM	FLASHER1	REV. 0.1
00002		OPT	NOF	
00003	0000 4F	CLR A		ACC NOW 0
00004	0001 B7 020F ALTER	STA A	\$020F	STORE ACC TO LIGHTS
00005	0004 CE 5500	LDX	#\$5500	*
00006	0007 09 WAIT	DEX		*WAIT
00007	0008 26 FD	BNE	WAIT	*
00008	000A 43	COM A		TOGGLE ACC
00009	000B 20 F4	BRA	ALTER	GO BACK TO RESTORE
00010		END		

Figure 10-35

Program to flash data LED's at regular interval.

## Procedure (continued)

9. Write a program to alternately store 1's and 0's to the display LED's. But this time, adjust the timing so the LED "on" time is longer than the "off" time. Then execute the program.

## Discussion

This program required two timing loops, to allow for the difference between on and off time. If your first program contained two timing loops of equal duration, it was a simple matter to modify the delay times. Figure 10-36 illustrates a second method for accomplishing the task. The delay times shown are for a **slow clock**. You may wish to change them if your Trainer has a **fast clock**.

In the next part of the experiment, you will add a decoder-driver and a common cathode, 7-segment display to the circuit.

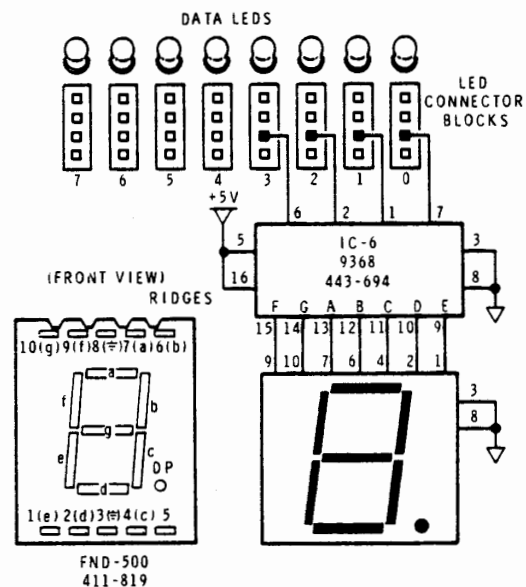
00001		NAM	FLASHER2 REV. 0.1	
00002		OPT	NOF	
00003	0000	ORG	0	
00004	0000 4F	CLR A		ACC NOW "0"
00005	0001 CE 5500	CYCLE LDX	#\$5500	LOGIC "0" TIME
00006	0004 B7 020F	STA A	\$020F	
00007	0007 43	COM A		BITS NOW "1"
00008	0008 09	HOLD1 DEX		
00009	0009 26 FD	BNE	HOLD1	
00010	000B CE FF00	LDX	#\$FF00	LOGIC "1" TIME
00011	000E B7 020F	STA A	\$020F	
00012	0011 43	COM A		BITS NOW "0"
00013	0012 09	HOLD2 DEX		
00014	0013 26 FD	BNE	HOLD2	
00015	0015 20 EA	BRA	CYCLE	ONE CYCLE COMPLETE
00016		END		

Figure 10-36  
Program to flash data LED's at a non-regular interval, with the on time longer than off time.

## Procedure (continued)

10. Switch the Trainer power off. Then, without disturbing the circuit wired to the Trainer, add the circuit shown in Figure 10-37. Use the large connector block affixed to the Trainer cabinet to hold the new circuit. The FND-500 (#411-819) display is the larger of the two displays.
11. Recheck your wiring, then switch the Trainer power on, and press RESET.
12. The lower four bits of your data byte will determine the digit displayed. Enter  $A5_{16}$  into address  $020F_{16}$ .
13. What is the bit pattern displayed by the lower four display LED's?  
— — — —2.
14. What is the hex equivalent?  $_{16}$ .
15. What is displayed by the new 7-segment display?  $_{16}$ .
16. Write a program that will cause the 7-segment display to count from 0 to  $F_{16}$  and then continuously repeat. Include a delay loop so that each digit will remain on long enough to be identified. Execute the program.

Figure 10-37  
Additional data display for first output  
circuit.



## Discussion

The circuit you just constructed contains a 4-line-to-7-segment decoder driver and a 7-segment, common cathode display. The decoder driver (9368) contains a large maze of combinational logic which allows it to decode four data bits and drive the proper segments in a 7-segment display to produce the corresponding hex digit.

The display circuit is a multiple LED array with common cathodes. The cathodes are grounded, and the decoder driver supplies the necessary power (approximately 30 mA at +5 volts) to light the selected LED segments.

If you had any questions concerning the program to increment the display, refer to Figure 10-38. It contains a simple program to increment the display from 0 to  $F_{16}$  at a slow rate. The simplicity of this program assignment removes the need to reset accumulator A after incrementing to  $0F_{16}$ . It continues beyond  $0F_{16}$ . But, since only the four lower bits of data are decoded, it appears to count to  $0F_{16}$  and then reset to  $00_{16}$ . Enter the program in Figure 10-38 and watch the eight data LED's. They show the actual value stored in accumulator A.

Next you will see how the MPU handles common-anode type displays. Also, you will see that a decoder driver is not necessary if you are willing to let the MPU do the decoding.

00001		NAM	STEP-UP	REV.0.4
00002		OPT	NOF	
00003	0000 4F	CLR A		START WITH 0
00004	0001 B7 020F UPDATE	STA A	\$020F	STORE TO OUTPUT
00005	0004 4C	INC A		ADD ONE
00006	0005 CE FFFF	LDX	#\$FFFF	TIME TO WAIT
00007	0008 09 UPDAT2	DEX		TIME RUNNING OUT
00008	0009 26 FD	BNE	UPDAT2	TIME UP YET?
00009	000B 20 F4	BRA	UPDATE	
00010		END		

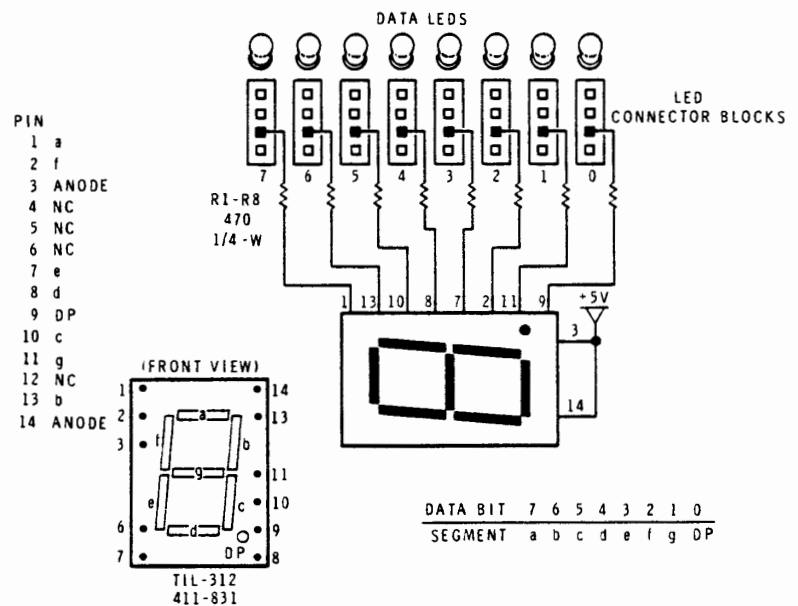
Figure 10-38

Program to increment the 7-segment display from 0 to  $F_{16}$  in an apparent cyclic manner.

## Procedure (continued)

17. Switch Trainer power off and remove the wires, decoder driver, and display package from the large connector block affixed to the Trainer cabinet.
18. Refer to Figure 10-39 and construct the circuit shown. Since the resistor leads are too short to reach from the connector block to the data LED connectors, insert the free end of each resistor into an unused connector socket. Then run hookup wire to the appropriate LED connector block.
19. Reexamine the circuit to make sure it is properly wired, and the resistor leads do not touch adjacent resistor leads. Then switch Trainer power on and press RESET.
20. This circuit, like the previous circuit, uses the address decoder and latches initially wired to the Trainer. Data stored at address  $020F_{16}$  will determine which display segment will light. Enter  $00_{16}$  at address  $020F_{16}$ . What does the display indicate?  $_{-16}$ .
21. Change the data to  $FF_{16}$ . What does the display indicate?  $_{-16}$ .
22. To light a particular segment in the display, the corresponding data bit must be logic 0. The table below the circuit in Figure 10-39 indicates the segments connected to the data bits. What bit pattern will produce the number 1 in the display?  $_{-2}$ .

Figure 10-39  
Additional data display.





23. Convert the bit pattern from step 22 to hex and enter it at address 020F<sub>16</sub>. Although it is possible to display two 1's, the correct 1 is produced when segments b and c are lit.
24. Load and execute the program shown in Figure 10-40. If your Trainer has a **fast clock**, you may want to change the contents of address 000B to provide a longer delay.

```

00001          NAM      CHAROUT1 REV. 0.1
00002          OPT      NOP
00003      020F      DISPLA EQU      $020F
00004 0000          ORG      0
00005 0000 CE 001A RECYCL LDX      #CODES      START OF TABLE
00006 0003 A6 00      NXTDIG LDA A      X          LOAD BIT PATTERN
00007 0005 B7 020F      STA A      DISPLA      STORE TO DISPLA
00008 0008 86 FF          LDA A      #$FF      *
00009 000A C6 55      HOLD1 LDA B      #$55      *
00010 000C 5A          HOLD2 DEC B          * WAIT
00011 000D 26 FD          BNE          HOLD2      *
00012 000F 4A          DEC A          *
00013 0010 26 F8          BNE          HOLD1      *
00014 0012 08          INX          POINT TO NXT PATTERN
00015 0013 8C 002A      CPX      #FINAL+1 LAST ONE YET?
00016 0016 27 E8          BEQ      RECYCL      IF SO, START AGAIN
00017 0018 20 E9          BRA      NXTDIG      IF NOT, NXT PATTERN
00018 001A 03          CODES FCB      $03,$9F,$25,$0D,$99,$49
          001B 9F
          001C 25
          001D 0D
          001E 99
          001F 49
00019 0020 41          FCB      $41,$1F,$01,$19,$11,$C0
          0021 1F
          0022 01
          0023 19
          0024 11
          0025 C0
00020 0026 63          FCB      $63,$85,$61
          0027 85
          0028 61
00021 0029 71          FINAL FCB      $71
00022          END

```

Figure 10-40

Program for incrementing the  
 7-segment display from 0 to F<sub>16</sub> in a  
 cyclic manner.

## Discussion

In this experiment, you have successfully eliminated a decoder driver, but at the expense of increased software. The program sequentially stores bit patterns to the display to make it appear as number 0 thru  $F_{16}$  are being stored.

Addresses  $001A_{16}$  thru  $0029_{16}$  contain the sixteen display codes in numerical sequence. This "look-up" table is then accessed by the index register to obtain the required code.

You may have noticed that the  $B_{16}$  digit had a decimal point lit next to it. This is sometimes used to indicate it is a B rather than a 6. If you prefer not to have the decimal point, you can change address  $0025_{16}$  to  $C1_{16}$ .

The display used in this circuit is of the common anode type, with the anodes connected to +5 volts. To turn on a segment, its cathode must be grounded. Therefore, a logic 0 turns on a segment while a logic 1 turns it off.

In some applications, it is convenient to assign each segment of the display its own address. In the next part of the experiment, you will see how this is accomplished.

## Procedure (continued)

25. Switch the Trainer power off. Then remove all of the wires and components from both large connector blocks.
  26. Refer to Figure 10-41 and construct this circuit on the Trainer's large connector block.
  27. Switch the Trainer power on. Then enter  $00_{16}$  at address  $02F0_{16}$ . Since only the  $D_0$  bit is connected to the display circuit, a logic 0 will turn a display segment on, and a logic 1 will turn the segment off.
  28. Advance the address and enter  $00_{16}$ . Continue this process and observe the display. Stop after you enter  $00_{16}$  at  $02F7_{16}$ . Notice that all of the display segments are lit, including the decimal point.
  29. Examine address  $02F0_{16}$  and watch the display. Now advance through the next seven address locations while you watch the display. What is finally displayed? \_\_\_\_\_. Why? \_\_\_\_\_
-

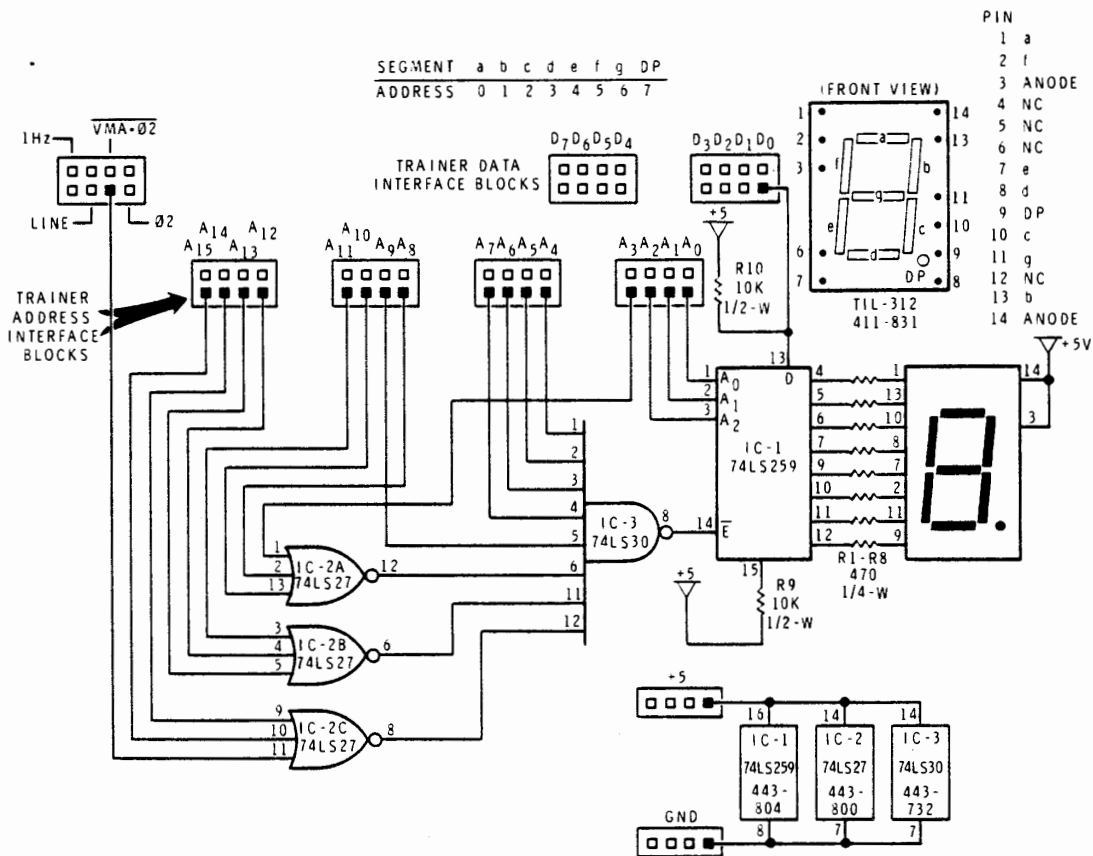


Figure 10-41

Circuit diagram of the fourth part of the output experiment.

A <sub>15</sub>	A <sub>12</sub>	A <sub>11</sub>	A <sub>8</sub>	A <sub>7</sub>	A <sub>4</sub>	A <sub>3</sub>	A <sub>0</sub>
0	0	0	1	0	1	1	1
0	0	0	1	0	1	1	1
0	0	0	1	0	1	1	1
0	0	0	1	0	1	1	1
0	0	0	1	0	1	1	1
0	0	0	1	0	1	1	1
0	0	0	1	0	1	1	1

Figure 10-42

Decoding chart for the fourth output circuit.

## Discussion

Each display segment now has its own address in memory. This is shown in the circuit decoding chart in Figure 10-42. Refer to Figure 10-41. Address bits A<sub>0</sub>, A<sub>1</sub>, and A<sub>2</sub> are decoded to select 1-of-8 bistable latches in IC1. Then during an MPU write operation, the logic information supplied by data bit D<sub>0</sub> is coupled into the selected latch. A logic 0 will turn on the appropriate display segment, while a logic 1 will turn off the segment. A table showing address/segment data is provide in Figure 10-41, above the circuit diagram. The remaining address bits, and  $\overline{\text{VMA}}\cdot\overline{\text{O2}}$  are used to enable ( $\overline{\text{E}}$ ) the latches.

Since this circuit is the write-only type, the D<sub>0</sub> line will "float" during an MPU read operation. Therefore, the D input of IC1 will go high (10 k ohm pull-up to +5 volts) and couple a logic 1 into the latch. This is why a segment went out when you examined its address without entering data.

## Procedure (continued)

DIGIT	DATA
0	03
1	9F
2	25
3	0D
4	99
5	49
6	41
7	1F
8	01
9	19
A	11
B	CO
C	63
D	85
E	61
F	71

Figure 10-44

Display data table for the character output program.

30. Load the program listed in Figure 10-43. Begin at address 0001<sub>16</sub>. (Address 0000<sub>16</sub> is reserved for data.) Notice that the comment column in step 0013<sub>10</sub> indicates this program will be used as a subroutine in a future program.
31. Refer to the display data table in Figure 10-44 and select a hex digit. Then enter the corresponding data at address 0000<sub>16</sub>.
32. Execute the program beginning at address 0001<sub>16</sub>. The hex digit you selected will appear in the 7-segment display.
33. Load the program listed in Figure 10-45. Begin at address 0102<sub>16</sub> (step 00006<sub>10</sub>). (Addresses 0100 and 0101<sub>16</sub> are reserved for data.) Then enter 39<sub>16</sub> at address 000F<sub>16</sub>. The program located at addresses 0001 thru 000F<sub>16</sub> is a subroutine for the program you just entered. The data stored at addresses 0124 thru 0133<sub>16</sub> serve as a look-up table for the 16 hex digits you will display. If your Trainer has a **fast clock**, you may want to change the contents of address 0115 to provide a longer delay.
34. Execute the program beginning at address 0102<sub>16</sub>. The 7-segment display will sequentially show the hex digits 0 thru F in a cyclic manner.

```

00001          NAM      CHAROUT2 REV. 0.2
00002          OPT      NOP
00003 0000      ORG      0
00004          SEGMENT EQU  $02F0
00005 0000 0001  CHARAC RMB  1
00006 0001 CE 02F7 OUTCHR LDX  #SEGMENT+7  TOP OF SEG. LIST
00007 0004 D6 00      LDA B  CHARAC  GET PATTERN
00008 0006 E7 00  NXTSEG STA B  0,X      STORE TO LATCH
00009 0008 56          ROR B              SHFT FOR NXT BIT
00010 0009 09          DEX
00011 000A 8C 02EF     CPX  #SEGMENT-1  LAST SEG. YET?
00012 000D 26 F7      BNE  NXTSEG
00013 000F 3E          WAI                  DONE(39 FOR SUBROUTINE)
00014          END

```

Figure 10-43

Program for writing data into a 7-segment display.

```
00001          NAM      OUTSTRIG REV. 0.3
00002 0100          ORG      $0100
00003          0000      CHARAC EQU      00
00004 0100 0002      ISAVE RMB      2
00005          0001      OUTCHR EQU     01
00006 0102 CE 0124  START LDX      #CODES  POINT TO CODE TBL
00007 0105 A6 00  NXTDIG LDA A      0,X    GET PATTERN
00008 0107 97 00          STA A  CHARAC  STORE IT
00009 0109 FF 0100      STX      ISAVE  SAVE INDEX
00010 010C BD 0001      JSR      OUTCHR  OUTPUT DIGIT
00011 010F FE 0100      LDX      ISAVE  RESTORE INDEX
00012 0112 86 FF          LDA A  #$FF    *
00013 0114 C6 55  HOLD1  LDA B  #$55    *
00014 0116 5A          HOLD2 DEC B      *
00015 0117 26 FD          BNE      HOLD2  * WAIT
00016 0119 4A          DEC A          *
00017 011A 26 F8          BNE      HOLD1  *
00018 011C 08          INX          POINT TO NXT CODE
00019 011D 8C 0134      CFX      #FINAL+1
00020 0120 27 E0          BEQ      START  RECYCLE
00021 0122 20 E1          BRA      NXTDIG  GET NXT DIGIT
00022 0124 03          CODES FCB      $03,$9F,$25,$0D,$99
        0125 9F
        0126 25
        0127 0D
        0128 99
00023 0129 49          FCB      $49,$41,$1F,$01,$19
        012A 41
        012B 1F
        012C 01
        012D 19
00024 012E 11          FCB      $11,$C0,$63,$85,$61
        012F C0
        0130 63
        0131 85
        0132 61
00025 0133 71          FINAL FCB      $71
00026          END
```

Figure 10-45

Program for outputting hex digits in sequence and in a cyclic manner. Requires program from Figure 10-43 as a subroutine.

## Discussion

With each reduction in hardware, there is generally an increase in support software. The program in Figure 10-43 is used only to output the necessary data bits to produce a single hex character. Since eight separate address locations are needed to light the 7-digit segments and the decimal point, the program must output eight bytes of data in order to produce the desired display. This is accomplished by using the index register to monitor each segment address and outputting the appropriate data from the B accumulator.

Remember that only the  $D_0$  data bit is connected to the display latch. Thus, you can enter the appropriate 8-bit word (for the hex digit) into the B accumulator and then write the word to the display, which only accepts the  $D_0$  bit. After the word is written, the B accumulator is rotated right, which places the next most significant data bit (for the hex digit) at the  $D_0$  position. The index register decrements to the next segment address and the program branches back to the store B accumulator step. This process continues until all of the display latches are filled, then the branch step defaults and the MPU goes into a wait for interrupt condition. The second program you entered is similar to the previous cyclic character output programs. At step  $00010_{10}$ , a jump to subroutine instruction sends the MPU back to the character output subroutine.

## Procedure (continued)

35. Switch the Trainer power off. Then remove the hookup wire and the components from the large connector block.
36. This completes this experiment. Return to the Unit Activity Guide of Unit 7.

## Experiment 5

### DATA INPUT

#### OBJECTIVES:

*Show how to construct a circuit for writing data to the microprocessor.*

*Demonstrate various methods for programming the microprocessor to accept externally applied data.*

*Demonstrate a software routine for debouncing a switch.*

*Show how to select a debounce routine to fit a specific system.*

### Introduction

Experiment 4 introduced you to various methods of outputting data from the microprocessor. In this experiment, you will learn how to input data. While many devices can be used to transfer data to a microprocessor (teletypewriter, tape reader, modem, transducer, etc.), they all accomplish their task in basically the same manner. You will use the Trainer binary data switches and four external pushbutton switches for data entry.

### Materials Required

- 1 ET-3400 Microprocessor Trainer
- 1 #1 switch
- 1 #2 switch
- 1 #3 switch
- 1 #4 switch
- 1 7400 integrated circuit (443-1)
- 1 74126 integrated circuit (443-717)
- 2 74LS30 integrated circuits (443-732)
- 1 74LS27 integrated circuit (443-800)

Hookup wire

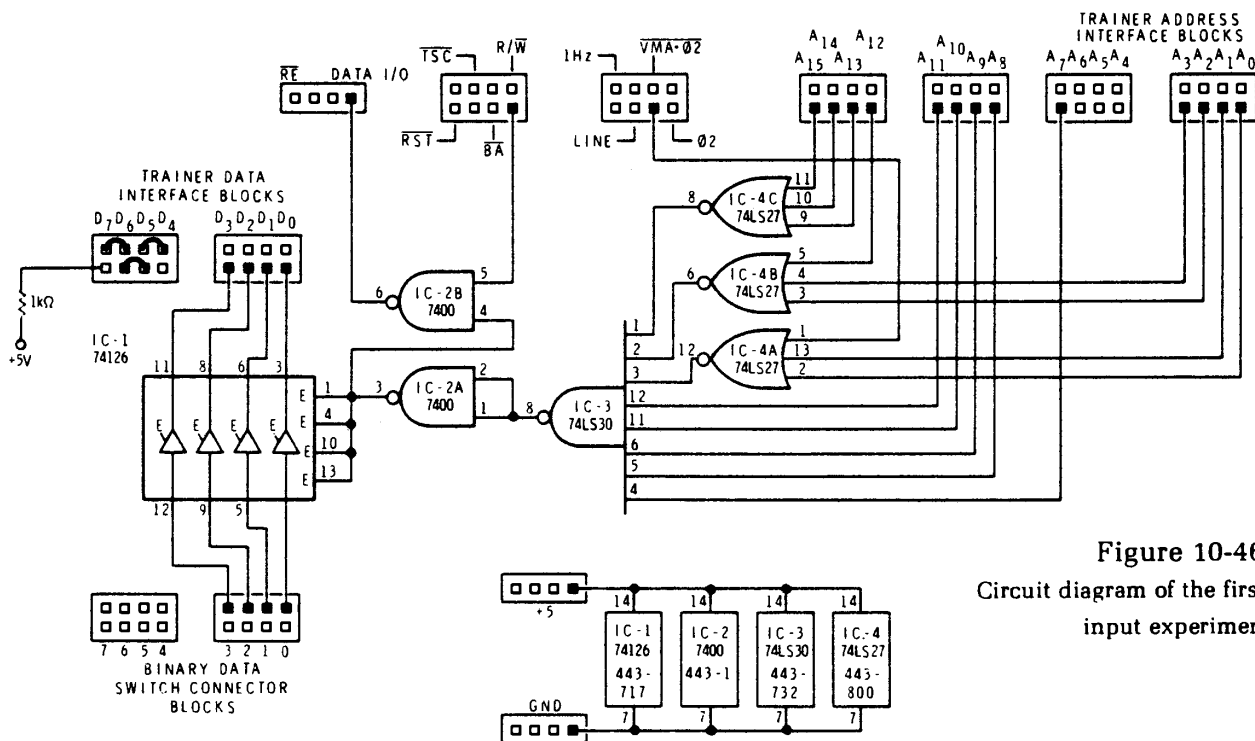


Figure 10-46

Circuit diagram of the first part of the input experiment.

## Procedure

1. In the first part of this experiment, you will interface four slide switches to the data bus of the MPU. Make sure the Trainer power is switched off. Then construct the circuit shown in Figure 10-46.
2. Make sure all of the binary data switches are down (logic 0). Then position switch 0 up to logic 1.

**NOTE:** You may have noticed that the display is faintly illuminated with Trainer power off. This is caused by current from the data lines being coupled through IC1 to the +5-volt connector block, and from there to the displays. Disregard the display with power off.

3. Switch Trainer power on and enter the program listed in Figure 10-47. Then execute the program beginning at address 0000<sub>16</sub>.

Figure 10-47

Program for inputting data from the binary data switches.

00001		NAM	INPUT-01	REV. 0.2
00002		OPT	NOF	
00003	0000 B6 0F80	LDA	A \$0F80	GET DATA
00004	0003 B7 0100	STA	A \$0100	SAVE IT
00005	0006 3E	WAI		DONE
00006		END		



4. Examine address 0100<sub>16</sub>. What is the contents? — —<sub>16</sub>.
5. Position data switch 0 down to logic 0. Then position data switch 1 up to logic 1.
6. Execute the program. Then examine address 0100<sub>16</sub>. What is the contents? — —<sub>16</sub>.
7. Position data switches 0 thru 3 up to logic 1.
8. Execute the program. Then examine address 0100<sub>16</sub>. What is the contents? — —<sub>16</sub>.
9. Enter the program listed in Figure 10-48.
10. Execute the program beginning at address 0000<sub>16</sub>. Now flip data switch 0 between logic 1 and logic 0 a number of times and observe the decimal point of Trainer display H. Notice that the decimal point is lit for a logic 1 and off for logic 0.

## Discussion

Refer again to the circuit in Figure 10-46. It is quite similar to the one used for outputting data. However, it operates like **read only memory**, with its data being influenced by external sources, (the "outside world").

00001	NAM	INPUT-02	REV. 0.2
00002	OPT	NOF	
00003 0000 B6 0F80 REDO	LDA A	\$0F80	GET DATA
00004 0003 B7 C167	STA A	\$C167	STORE IT
00005 0006 20 F8	BRA	REDO	GO BACK AGAIN
00006	END		

Figure 10-48  
Program to follow and display input  
from data switch 0.

The circuit is partially decoded as shown in Figure 10-49. When any of the specified addresses is selected, the buffer drivers of IC1 are enabled through inverter IC2A. This allows the data switch logic to be coupled to the Trainer data bus buffers. As soon as the  $R/\overline{W}$  line goes high (MPU read), gate IC2B enables the input portion of the Trainer data bus buffers through the  $\overline{RE}$  line.

You may have noticed one flaw in the circuit. The buffers in IC1 are always enabled when any of the circuit decoded addresses are selected. Therefore, it is important that you as the programmer do not try to write to these addresses. If you did so, the buffers would try to source or sink the data lines and result in possible circuit damage. One way to avoid this problem is to disconnect pin 4 of IC3 from  $A_7$  and connect it to the  $R/W$  line. This will disable IC1 during an MPU write, but the circuit address coding is now changed to  $00001111\cdots0000_{16}$ .

Both programs in this experiment used address  $0F80_{16}$  as an input port. The first retrieves data from  $0F80_{16}$  and stores it at  $0100_{16}$ .

The second program also retrieves data from  $0F80_{16}$ . But this time, it is stored at  $C167_{16}$ , the address of the decimal point for Trainer display H. Since only the  $D_0$  data bit is connected to the Trainer display, data switch 0 is the only switch to affect the display. The program continuously branches back and retrieves switch data immediately after storing the previous data. Thus, when you changed the position of data switch 0, the decimal point appeared to follow the logic value of the changing switch position.

Next, some additional hardware and software features will be added to the circuit.

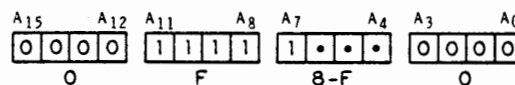


Figure 10-49  
Decoding chart for the first input circuit.

## Procedure (continued)

11. Refer to Figure 10-50 and construct the circuit shown. This circuit interconnects with the first circuit you constructed. Remember, the pushbutton pins are fragile. Press straight down when you install them in the large connector block, mounted on the Trainer cabinet.
12. Position all of the Trainer binary data switches up to logic 1.
13. Load the program listed in Figure 10-51, beginning at address 0000<sub>16</sub> (program step 00007).

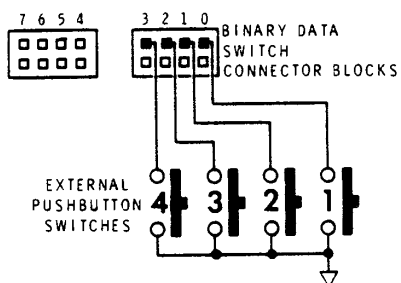


Figure 10-50

Added circuitry for the data input experiment.

00001		NAM	KEYINDIC REV.0.2
00002		OPT	NOF
00003	0000	ORG	0
00004	FE3A	OUTCH	EQU \$FE3A
00005	FE52	OUTSTR	EQU \$FE52
00006	FCBC	REDIS	EQU \$FCBC
00007	0000 BD 0031	ONE	JSR CLRDIS CLEAR DISPLAY ROUTINE
00008	0003 BD FCBC		JSR REDIS RESET DIGIT POSITION (LEFT)
00009	0006 F6 0F80		LDA B \$0F80 LOOKING FOR KEY CLOSURE
00010	0009 86 30		LDA A #\$30 BIT PATTERN FOR #1
00011	000B 56		ROR B MOVES "D0" BIT TO C REGISTER
00012	000C 25 02		BCS TWO NOT ONE? GO TO TWO
00013	000E 8D 17		BSR XECUTE OUTPUT A #1
00014	0010 86 6D	TWO	LDA A #\$6D BIT PATTERN FOR #2
00015	0012 56		ROR B MOVES "D1" BIT TO C REGISTER
00016	0013 25 02		BCS THREE NOT TWO? GO TO THREE
00017	0015 8D 10		BSR XECUTE OUTPUT A #2
00018	0017 86 79	THREE	LDA A #\$79 BIT PATTERN FOR #3
00019	0019 56		ROR B MOVES "D2" BIT TO C REGISTER
00020	001A 25 02		BCS FOUR NOT THREE? GO TO FOUR
00021	001C 8D 09		BSR XECUTE OUTPUT A #3
00022	001E 86 33	FOUR	LDA A #\$33 BIT PATTERN FOR #4
00023	0020 56		ROR B MOVES "D3" BIT TO C REGISTER
00024	0021 25 DD		BCS ONE NOT FOUR? GO BACK TO ONE
00025	0023 8D 02		BSR XECUTE OUTPUT A #4
00026	0025 20 D9		BRA ONE RETURN, RECHECK FOR CLOSURE
00027	0027 BD FE3A	XECUTE	JSR OUTCH MONITOR ROUTINE OUTPUTS CHAR.
00028	002A CE 0100		LIX #\$0100 ENTER TIMING LOOP
00029	002D 09	HOLD	DEX TIME RUNNING OUT
00030	002E 26 FD		BNE HOLD TIME OUT YET?
00031	0030 39		RTS RETURN, RECHECK FOR CLOSURE
00032	0031 BD FE52	CLRDIS	JSR OUTSTR THE FOLLOWING CLEARS DISPLAY
00033	0034 00	FCB	00,00,00,00,00,\$80
	0035 00		
	0036 00		
	0037 00		
	0038 00		
	0039 80		
00034	003A 39	RTS	
00035		END	

Figure 10-51

Program to display the pushbutton numbers in a sequential manner.

14. Execute the program beginning at address  $0000_{16}$ . The decimal point in display C will light to show the program is working.
15. Press one of the four pushbuttons and note the displayed result.
16. Simultaneously press any two pushbuttons and note the result.
17. Simultaneously press any three pushbuttons and note the result.
18. Simultaneously press all four pushbuttons and note the result.

## Discussion

The four pushbuttons in this experiment simply provide a convenient substitute for the four Trainer data switches. You could obtain the same result by manipulating the data switches. However, the pushbuttons will be needed in the next portion of the experiment.

The program shown in Figure 10-51 makes extensive use of the Trainer monitor routines located in ROM. These include OUTCH, OUTSTR, and REDIS. An earlier programming experiment showed how to use these routines.

Recall that OUTCH outputs a 7-segment code from accumulator A to the display indicated by a display pointer. OUTSTR outputs a string of characters to the displays. REDIS resets the display pointer so that the first character displayed by OUTCH or OUTSTR is in display H.

In addition, the program has two subroutines of its own. CLRDIS (for clear displays) is in addresses  $0031_{16}$  through  $003A_{16}$ . It uses OUTSTR to clear the six displays. XECUTE is in addresses  $0023_{16}$  through  $0030_{16}$ . It uses OUTCH to display a character and then goes into a short delay.

The program starts at address  $0000_{16}$ . The first instruction jumps the MPU to the CLRDIS subroutine. After the displays are cleared, the MPU returns to the instruction at address  $0003_{16}$ . This instruction directs the MPU to the REDIS subroutine. This sets the display pointer to display H.

The MPU returns to the instruction in address  $0006_{16}$ . Pushbutton data is now loaded into the B accumulator. Next, the 7-segment pattern for a "1" is loaded in the A accumulator. The  $D_0$  bit in the B accumulator is examined. If it is a one (#1 pushbutton not pressed), the program branches forward to TWO. If the  $D_0$  bit is a zero, the program branches to XECUTE. XECUTE displays the 1 in display H.

After XECUTE, an RTS sends the program back to address  $0010_{16}$ . The A accumulator is loaded with the bit pattern for the digit "2". Then the B accumulator is again rolled right to test for a #2 pushbutton actuation. If #2 was pushed, it will be displayed; otherwise, the program will advance and test the remaining pushbuttons.

The pushbuttons that test true determine the numbers displayed. However, the display pointer determines the display that contains the number.

After all of the pushbuttons have been tested, the display is cleared and the display pointer again points to display H.

In many applications, the MPU constantly scans the input switches looking for input data. However, in some applications this would waste too much of the MPU's time. A better approach is to let the MPU ignore the keyboard until a key is depressed. This is possible through the use of interrupts. In the next part of the experiment you will see how a keyboard can control the MPU through the interrupt line. You will also see how a debounce subroutine works.

## Procedure (continued)

19. Switch the Trainer power off. Then refer to Figure 10-52 and add the circuit shown to the circuit already wired to the Trainer. There should be enough room near the left end of the large connector block "on board" the Trainer to hold the additional 74LS30. Notice that the inputs to the 74LS30 are connected in parallel with the data lines leaving the four pushbutton switches. IC2C is one of the unused gates in IC2.

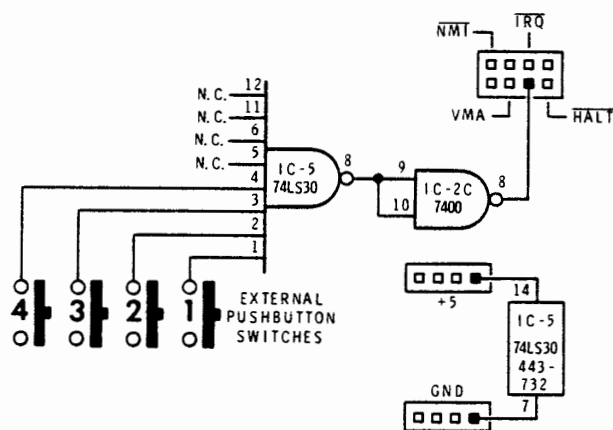


Figure 10-52  
Interrupt circuitry for data input ex-  
periment circuit.

20. Switch Trainer power on. Then refer to Figure 10-53 and enter the program listed beginning at address 0000<sub>16</sub>. Notice that after you enter the 3B<sub>16</sub> at address 002B<sub>16</sub>, you must go to address 00F7<sub>16</sub> to enter the remaining data. 002C and 002D<sub>16</sub> are temporary registers.
21. Now enter 00<sub>16</sub> into address 0100 thru 0110<sub>16</sub>. These addresses are used as data storage registers.
22. Execute the program beginning at address 0000<sub>16</sub>. The display will go blank.
23. Strike each pushbutton sequentially in a 1, 2, 3, 4 order. When you strike each button, use a moderate force, such as you would use when typing with a mechanical typewriter. The data you entered is stored in memory and will not be displayed.

```

00001          NAM      DBOUNCE1 REV. 0.4
00002          OPT      NOP
00003      0F80      INPUT EQU      $0F80
00004 0000 0E          CLI          READY FOR INTERRUPT
00005 0001 CE 0100 PROGRA LDX      $$0100 POINT TO STORAGE
00006 0004 01          NOP          *
00007 0005 01          NOP          * LOCATION FOR PROGRAM
00008 0006 20 F9      BRA      PROGRA *
00009 0008 B6 0F80 GETDAT LDA A INPUT GET DATA
00010 000B B1 002C      CMP A TEMP  IS IT LIKE BEFORE?
00011 000E 27 07      BEQ      SAME  IF SO, GO TO SAME
00012 0010 B7 002C      STA A TEMP  IF NOT, STORE IN TEMP
00013 0013 7F 002D      CLR      COUNT RESET COUNTER TO ZERO
00014 0016 3B          RTI
00015 0017 C6 40      SAME LDA B $$40 NUMBER OF CHECKS
00016 0019 F1 002D      CMP B COUNT ENOUGH CHECKS YET?
00017 001C 27 04      BEQ      LEGAL IF SO, GO TO LEGAL
00018 001E 7C 002D      INC      COUNT IF NOT, INCREMENT COUNT
00019 0021 3B          RTI
00020 0022 43      LEGAL COM A      INVERT LOGIC
00021 0023 A7 00      STA A X      PLACE IN STORAGE
00022 0025 7F 002D      CLR      COUNT RESET COUNTER TO ZERO
00023 0028 08          INX          POINT TO NEXT STORAGE PLACE
00024 0029 DF 02      STX      PROGRA+1 SAVE I DURING RTI
00025 002B 3B          RTI
00026 002C 0001      TEMP RMB      1
00027 002D 0001      COUNT RMB     1
00028 00F7          ORG      $00F7 INTERRUPT VECTOR
00029 00F7 7E 000B      JMP      GETDAT
00030          END

```

Figure 10-53

Program to software debounce the  
input pushbuttons.

24. Examine address  $0003_{16}$ . It should contain  $04_{16}$ , which is the number of pushbutton contact closures made. Change the contents back to  $00_{16}$ .
25. Examine addresses  $0100$  thru  $0103_{16}$ . They should contain 01, 02, 04, and  $08_{16}$  respectively. Change the data in these four locations back to  $00_{16}$ . Even though the pushbuttons are labeled 1, 2, 3, and 4, they are connected to data lines  $D_0$ ,  $D_1$ ,  $D_2$ , and  $D_3$ . Therefore, the switches will enter the binary values 1, 2, 4, and 8.
26. Execute the program. Then press each pushbutton twice in succession (1, 1, 2, 2, 3, 3, 4, 4). Address  $0003_{16}$  now contains  $08_{16}$ , representing eight pushbutton contact closures. Enter  $00_{16}$  at address  $0003_{16}$ .
27. Examine addresses  $0100$  thru  $0107_{16}$ . They will show the value of each pushbutton pressed and the sequence it was pressed. Change the data in these address back to  $00_{16}$ .
28. Examine address  $0018_{16}$ . It should contain data  $40_{16}$ . Change the value to  $00_{16}$ .
29. Execute the program. Then press each pushbutton once in sequence.
30. Examine address  $0003_{16}$  and record the contents.  $_{16}$ . This number should equal  $04_{16}$ . However, it may be higher.
31. Record the data in the following addresses. You need only examine the number of addresses that correspond to the value recorded in step 30.

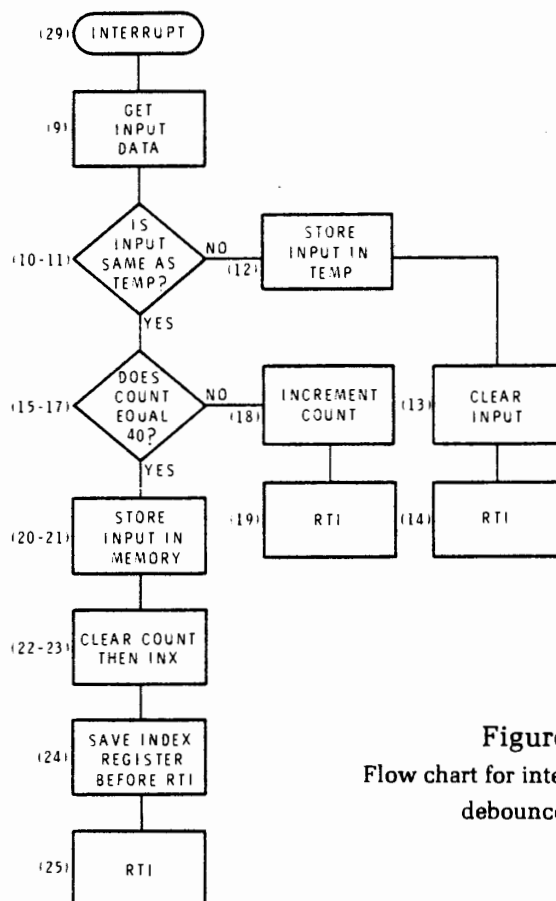
0100	--	0109	--
0101	--	010A	--
0102	--	010B	--
0103	--	010C	--
0104	--	010D	--
0105	--	010E	--
0106	--	010F	--
0107	--	0110	--
0108	--		

## Discussion

IC5 and gate IC2C provide an interface between the four external pushbuttons and the interrupt request line ( $\overline{\text{IRQ}}$ ). The remaining circuitry functions as before. Thus, whenever you attempt to enter data with the pushbutton switches, a request for program interrupt signal is sent to the microprocessor.

The program listed in Figure 10-53 processes the interrupt and debounces the keys. The program is actually two programs in one. The first part (steps 00005 through 00008) serves as a “simulated” program that runs in an eternal loop until it is interrupted. The remaining program steps actually service the input data pushbuttons during the interrupt. This is the program we will deal with.

Figure 10-54 is a flow chart for the interrupt program. The numbers at each block represent the assembled program steps.



**Figure 10-54**  
Flow chart for interrupt routine in the  
debounce program.



When the MPU receives an interrupt request; it completes the instruction it is presently performing, loads the internal registers and accumulators into the stack, sets the interrupt mask in the condition code register, then examines ROM to find out where the program counter is to be vectored. The vector address instruction sends the program counter to the beginning address of the interrupt program.

Pushbutton data is loaded and compared to the data in the temporary register (address  $002C_{16}$ ). Since this is the first time data is examined, there can be no match. Therefore, the pushbutton data is stored in the temporary register, the counter register (address  $002D_{16}$ ) is reset, and the MPU returns to the original program. This is the first time the MPU looks at the pushbuttons during the debounce routine. The data in the temporary register will serve as the reference for all future interrupts. If the input data changes, this new data will be entered, and the counter register will be reset. The counter is used later in the interrupt program to monitor the number of data examinations performed.

Upon return from the interrupt program, the MPU pulls the accumulator and register data from the stack. This clears the interrupt mask, and since you still have the pushbutton pressed, the MPU immediately acknowledges the interrupt request. Whereupon, it stores into the stack, sets the mask, and looks up the interrupt vector.

Pushbutton data is again compared with the temporary register. This time, it matches. Thus, allowing a branch to address  $0017_{16}$ . Data  $40_{16}$  is loaded into the B accumulator and then compared with the count register. Since the count is zero, there is no match. Therefore, the count is incremented and the MPU returns to the main program.

Assuming you are still holding the pushbutton down, the MPU goes through the interrupt routine 38 more times (39 total). During the 40<sup>th</sup> cycle, if the data is still good, the MPU will be satisfied that the data supplied by the pushbutton is true, and the program is allowed to branch to address  $0022_{16}$ .

The contents of accumulator A (pushbutton data) is complemented and stored at the address pointed to by the index register. This address was loaded into the index register in the main program. It is the first of  $17_{10}$  addresses you reserved for data when you performed the experiment.

The counter register is cleared (in case the same pushbutton is again pressed). The index register is incremented and stored at address  $0002_{16}$ . This points to the next data address, in preparation for the next pushbutton closure. Finally, the MPU returns to the main program.

You may have wondered why the pushbutton data was complemented before storage (address 0022<sub>16</sub>). This was necessary since the pushbuttons were wired using inverse logic. That is, when the #1 pushbutton was pressed, data 1111 1110<sub>2</sub> was transferred on the data bus, rather than 0000 0001<sub>2</sub>. Thus, it was necessary to invert the data for "logical" interpretation.

In the second part of this portion of the experiment, you changed the number of data examinations from 40<sub>16</sub> to 00<sub>16</sub> (address 0018<sub>16</sub>). Then when you entered four pushbutton closures, you probably found more than four entries stored at address 0003<sub>16</sub>. This occurred because the contacts of a switch tend to bounce open and closed a number of times before they stay closed. Since the bounce period can last many milliseconds, the MPU could treat each bounce as a separate entry, as you probably experienced.

Again look at the data you recorded in step 31. As you know, the program is designed to store one pushbutton closure in each address. A series of two or more identical entries indicates bounce. You may even have one or two zeroes recorded. This occurred because the contacts opened after an interrupt request, but before the data could be tested. Thus, a zero is stored.

Contact bounce can not be tolerated. But, what is a desirable number of switch samples? This will depend on the type of switch. If the sample is too low, bounce can occasionally get through. Large samples waste time and may require long switch hold-down periods. Normally five to eight samples are sufficient for a program of the type you used in this experiment. However, some switches will produce excessive bounce. As a precaution, 40 samples are used in the program.

Your Microprocessor Trainer uses a similar software routine for key debounce. This is stored in its ROM. Another method for debouncing a switch is to use cross-coupled NAND gates. They latch on the first closure and any additional bouncing is ignored. Regardless of the method used, you must debounce any mechanical switch used for data entry.

If you experiment with the sample rates in the program you entered, always be sure to change the data at addresses 0003<sub>16</sub> and 0100 through 0110<sub>16</sub> to 00<sub>16</sub> before you execute the program.

## **Procedure (continued)**

32. This completes this experiment. Switch the Trainer power off. Then remove all of the hookup wire and components from the two large connector blocks.

## Experiment 6

### INTRODUCTION TO THE PERIPHERAL INTERFACE ADAPTER (PIA)

#### OBJECTIVES:

*Show how to interface the MPU with the outside world using a PIA (6820).*

*Demonstrate various ways the PIA can be initialized as an input, output, or input/output (I/O) device.*

#### Introduction

As you have seen in the previous experiments, the need for latches and drivers to communicate with the MPU from the outside world can become quite burdensome. Then, once you have established a hardware interface circuit, you can not easily modify its function. However, the PIA can simplify your interface requirements in such a way that standardization is possible regardless of application. Therefore, you can easily develop interface systems compatible with your hardware and software needs. This is possible because most of the PIA performance characteristics are software controlled. Thus, performance and design features can be modified with little difficulty. In this experiment, some of the PIA's characteristics will be examined.

#### Material Required

- 1 ET-3400 Microprocessor Trainer
- 2 1000 ohm, 1/4-watt, 10% resistors
- 1 7400 integrated circuit (443-1)
- 1 74LS30 integrated circuit (443-732)
- 1 74LS27 integrated circuit (443-800)
- 1 6820 PIA integrated circuit (443-843)

Hookup wire



3. Switch the Trainer power on. Then enter the program listed in Figure 10-56.
4. Set all of the binary data switches to their down (logic 0) position. Then execute the program. The display will go blank.
5. Randomly set the data switches and observe the data LED's. Notice that the LED corresponding to each switch follows the logic level of the switch.
6. Change the instruction at address 0017<sub>16</sub> to 43<sub>16</sub>.
7. Execute the program and again randomly set the data switches. Notice that the data LED's now show the complement logic level of the switches.
8. Refer to Figure 10-56 and briefly describe the "service routine" section of the program. \_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_

```
00001          NAM      PIA-EXP1 REV. 0.2
00002          OPT      NOP
00003          *INITIALIZE PIA
00004 0000 86 00      LDA A  #00      0=INPUT
00005 0002 B7 8000      STA A  $8000  A SIDE NOW INPUT
00006 0005 86 04      LDA A  #04      SET TO COMMUN.
00007 0007 B7 8001      STA A  $8001
00008 000A 86 FF      LDA A  #$FF      1=OUTPUT
00009 000C B7 8002      STA A  $8002  B SIDE NOW OUTPUT
00010 000F 86 04      LDA A  #04      SET TO COMMUN.
00011 0011 B7 8003      STA A  $8003
00012          *SERVICE ROUTINE
00013 0014 B6 8000 RESERV LDA A  $8000  GET DATA
00014 0017 01          NOP
00015 0018 B7 8002      STA A  $8002  STORE TO OUTPUT
00016 001B 20 F7      BRA      RESERV  DO IT AGAIN
00017          END
```

Figure 10-56

Program to initialize and use the PIA  
for data input and output.

## Discussion

By now you are quite familiar with address decoding. Therefore, the discussion will deal with the PIA. Figure 10-57 is a decoding chart for the circuit you wired to the Trainer. If during this discussion you don't fully understand a specific function of the PIA, refer to the PIA section in Unit 8 and the PIA data sheet in Appendix B.

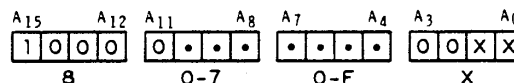


Figure 10-57

Decoding chart for the circuit in Figure 10-56.

Three chip-select pins on the PIA provide for easy decoding. They help eliminate address decoding gates. In small systems where partial address decoding can be tolerated, these three pins may be all that is needed to access the device. Notice that the PIA responds to addresses 8000<sub>16</sub> through 8003<sub>16</sub>.

The reset pin clears the PIA registers and is normally used at system turn-on. Therefore, it is connected to the system reset line.

The read/write pin controls data flow direction in the PIA in a manner similar to the RAM. Thus, it is connected to the R/W line from the MPU.

Interrupt request lines A and B can be wire OR'ed as in this experiment. Thus, each can transmit an MPU interrupt on the  $\overline{\text{NMI}}$  or  $\overline{\text{IRQ}}$  lines (in this experiment, the feature is not used).

Control pins A1 and B1 are inputs that are used to control the internal PIA interrupt flags. Control pins A2 and B2 can also serve as interrupt inputs or as peripheral control outputs. Since these features are not required for this experiment, each pin is pulled to a logic 1 to provide a termination and prevent undesired PIA interrupts.

The enable pin controls data transfer between the PIA and MPU. Since MPU data transfer occurs during time  $\phi 2$ , this pin is connected to Trainer  $\phi 2$ .

Data pins 0 through 7 are connected to the MPU data bus for device communication.

Peripheral pins A0 through A7 can be programmed as inputs or outputs. In this experiment, they are programmed as inputs and are connected to the binary data switches.

Peripheral pins B0 through B7 can also be programmed as inputs or outputs. In this experiment, they are programmed as outputs and are connected to the data LED's. Normally, the B side is used as an output because of its extra drive capabilities.

As you learned in Unit 8, the PIA must be initialized before it can function properly. This is accomplished through a software routine. Because initialization is accomplished by software, the PIA's operation can be modified at any time during the program.

When the PIA receives a reset pulse, its six memory accessible registers are cleared. Thus, whenever the Trainer RESET key is pressed, the PIA is reset. Because of this, the PIA must be initialized after each reset.

The program you entered (Figure 10-56) used the instructions in addresses 0000 through 0013<sub>16</sub> to initialize the PIA. This programs the A side of the PIA as an input. Then, 04<sub>16</sub> was loaded into control register A. This sets bit 2 of the control register high, which isolates the data direction register and accesses the output register.

In a like manner, the B side of the PIA is set up as an output by loading FF<sub>16</sub> into the data direction register. Then the data direction register is isolated and the output register accessed by loading 04<sub>16</sub> into the control register.

The remaining steps in the program comprise the service routine. The MPU reads data from the A side of the PIA and stores it to the B side. The "branch always" instruction holds the program in the service routine. Once the PIA is initialized, it will function as programmed until it is reset.

When you changed the instruction at address 0017<sub>16</sub> to 43<sub>16</sub>, you instructed the MPU to complement the data in the A accumulator before storing the data.

The program listed in Figure 10-58 is the same as the program you used, with one exception; the index register is used in place of the A accumulator for initializing the PIA. This reduced the number of program steps required.

### Procedure (continued)

9. Do not disassemble the circuit you have wired to the Trainer. It will be used in the next experiment. Proceed to Experiment 7.

```

00001                                NAM      PIA-EXP2 REV. 0.2
00002                                OPT      NOP
00003                                *INITIALIZE PIA
00004 0000 CE 0004                    LDX      #$0004
00005 0003 FF 8000                    STX      $8000
00006 0006 CE FF04                    LDX      #$FF04
00007 0009 FF 8002                    STX      $8002
00008                                *SERVICE ROUTINE
00009 000C B6 8000 RESERV LDA A      $8000      GET DATA
00010 000F 01                        NOP
00011 0010 B7 8002                    STA A      $8002      STORE TO OUTPUT
00012 0013 20 F7                      BRA      RESERV      DO IT AGAIN
00013                                END

```

Figure 10-58

Alternate program to initialize the PIA  
for data input/output.



## Experiment 7

### AUDIO OUTPUT

#### OBJECTIVES:

*Show how a transducer can be interfaced with an MPU.*

*Provide an opportunity to write an output program that will supply the data to drive a speaker.*

*Demonstrate how different audible tones can be generated.*

### Introduction

With the proper interface, microprocessors are capable of producing meaningful audio sounds. These signals are often useful as indicators when the operator cannot monitor the display and would like to know when an event has occurred.

Audible sounds can be produced in two ways. The first simply uses a buzzer that is activated by a change in output logic level, in the same manner as an LED. The second method actually drives an audio speaker. This experiment will use the second method to produce a variety of meaningful tones.

### Materials Required

- 1 ET-3400 Microprocessor Trainer with PIA circuit wired to the large connector block
- 2 100 ohm, 1/2-watt, 10% resistors
- 1 100  $\mu$ F electrolytic capacitor
- 1 Speaker
- Foam tape (from previous experiment)
- Solder (from Trainer kit)
- Hookup wire

## Procedure

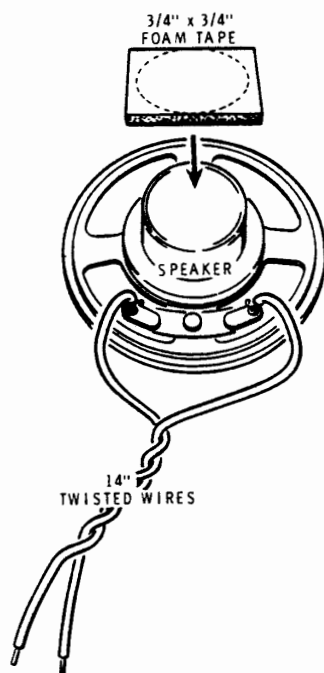


Figure 10-59  
Speaker preparation.

1. Cut two 14" hookup wires and remove 1/4" of insulation from the ends of each wire. Then twist the wires together, leaving about 2" untwisted at each end.
2. Remove the speaker from its packing container. Then refer to Figure 10-59 and solder the two wires at one end of the 14" twisted wire pair to the two speaker terminals. Disregard any polarity marks on the speaker.
3. Cut a 3/4"  $\times$  3/4" piece of foam tape. Remove the paper backing from one side and press the tape onto the end of the magnet on the speaker. Then remove the paper backing from the other side of the tape and affix the speaker to the sloping back of the Trainer cabinet near the Power switch. Position the speaker lugs up away from the Trainer.
4. Switch the Trainer power off. Remove the eight wires interconnecting the data LED's and PIA. Then remove the eight wires interconnecting the binary data switches and PIA.
5. Refer to Figure 10-60 and construct the circuit shown. Connect the speaker wires and the capacitor to the unused large connector block. (Additional components will be added later in the experiment.) The gate is part of IC3 in the original circuit. You can use pin 7 of IC3 for speaker ground. Figure 10-61 shows the complete circuit wired to your Trainer.

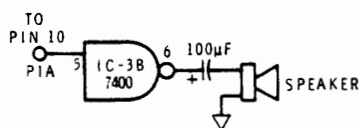
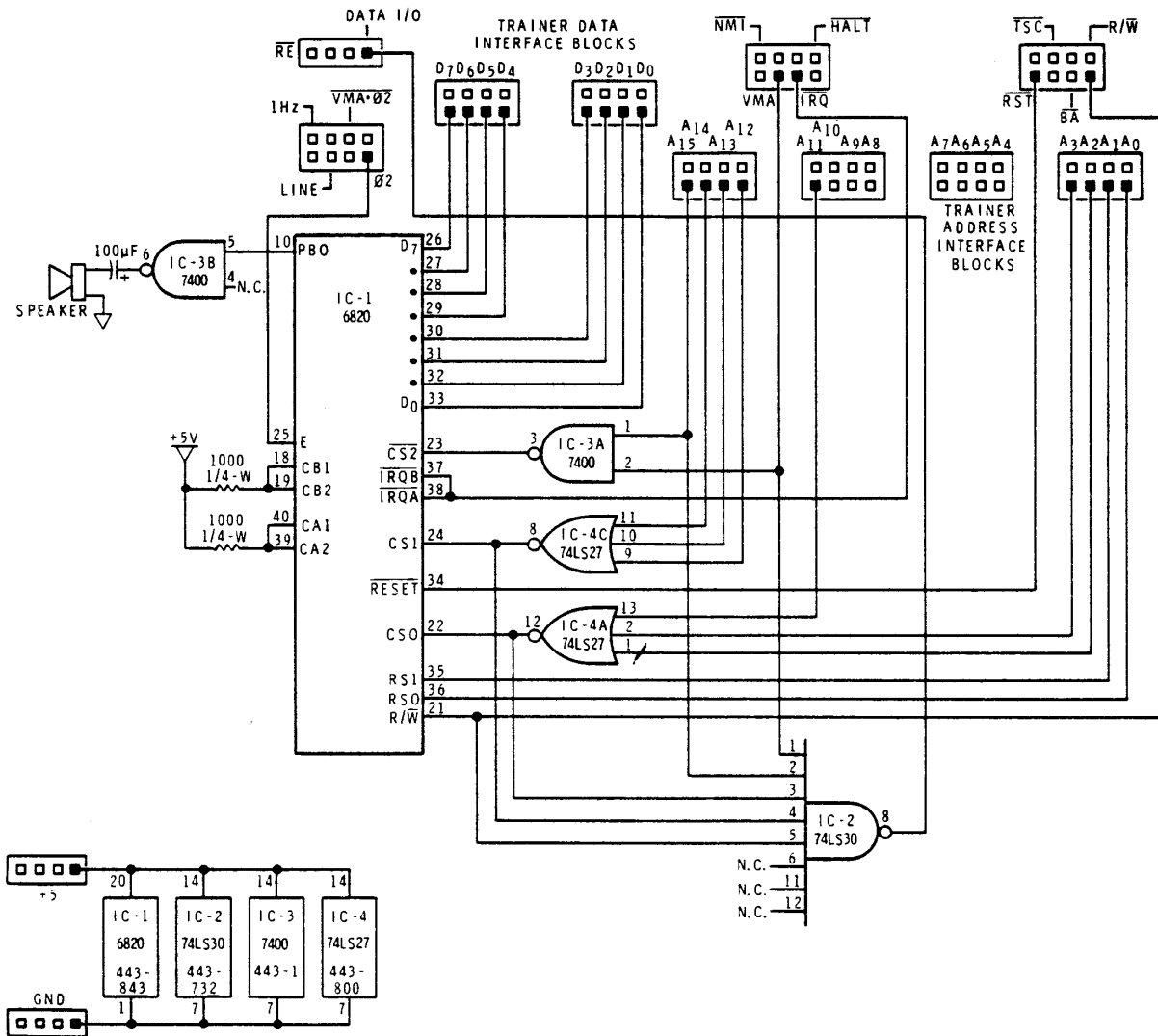


Figure 10-60  
Speaker/PIA interface circuit.





**Figure 10-61**  
Circuit diagram for the audio output  
experiment.

6. Switch the Trainer power on. Load the program listed in Figure 10-62. Begin at address  $0003_{16}$  (line 00008). Notice that a number of program steps have no data entry. Also, lines  $00027_{16}$  and  $00052_{16}$  contain assembler equate statements and should be ignored. After you enter  $20_{16}$  at address  $004B_{16}$ , go to address  $00F7_{16}$  to enter  $3B_{16}$ .
7. Press RESET, then install a hookup wire between LINE and  $\overline{\text{IRQ}}$ . This wire is not shown in the circuit diagram.
8. The program you entered is for a clock function. Addresses 0000, 0001, and  $0002_{16}$  are reserved for the seconds, minutes, and hours of the clock respectively. Enter the desired time into these three addresses.
9. Execute the program beginning at address  $0003_{16}$ . Each time the seconds count updates, you should hear a "tick" from the speaker.

## Discussion

### HARDWARE

Gate IC3B supplies the current needed to drive the speaker, while the  $100\ \mu\text{F}$  capacitor protects the gate. If the program stopped during a logic high output, the speaker would act as a direct short to ground. The capacitor coupling prevents this possibility. A side benefit of the capacitor is the RC time constant it forms with the internal gate circuitry. The pulse width for logic 1 and logic 0 transitions is different, producing a "tick-tock" sound.

### SOFTWARE

The clock program you entered is very similar to the clock program from Experiment 2. Two instructions were added to initialize the PIA (lines 00008 and 00009). Also, a store instruction was added to the 60-second timer subroutine (line 00022).

The store instruction outputs the seconds digit information to the PIA (B side) every time the digit increments. Since the  $D_0$  data bit is the only bit that changes during each time update, only peripheral output PB0 (pin 10) is needed to supply speaker data.

If you have any questions concerning the clock program, refer to Experiment 2.

```

00001      NAM      CLOCK-3 * REV 0.2
00002      **LINE  ACCURACY CLOCK PROGRAM
00003      OPT      NOP
00004 0000 0001  SECOND RMB 1
00005 0001 0001  MINUTE RMB 1
00006 0002 0001  HOURS  RMB 1
00007      ** PIA  INITIALIZATION
00008 0003 CE FF04  LDX  #$FF04
00009 0006 FF 8002  STX  $8002
00010      ** INTERRUPT HANDLING
00011 0009 CE 003D  TIMPAS LDX  #$003D 61
00012 000C 09      ONE60T DEX  TIME TICKING OFF
00013 000D 27 04  BEQ  TIMEUP 60 PULSES YET?
00014 000F 0E      CLI
00015 0010 3E      WAI  WAITING
00016 0011 20 F9  BRA  ONE60T GO BACK & WAIT AGAIN!
00017      ** INCR ONE SECOND AND UPDATE
00018 0013 C6 60  TIMEUP LDA B  #$60  SIXTY SECONDS,SIXTY MINUTES
00019 0015 0D      SEC  ALWAYS INCREMENT SECONDS
00020 0016 8D 16  BSR  INCR  INCREMENT SECONDS
00021 0018 96 00  LDA  A  SECOND
00022 001A B7 8002  STA  A  $8002
00023 001D 8D 0F  BSR  INCR  INCREMENT MINUTES IF NEEDED
00024 001F C6 13  LDA  B  #$13  TWELVE HOUR CLOCK
00025 0021 8D 0B  BSR  INCR  INCREMENT HOURS
00026 0023 BD FCBC  JSR  REDIS  RESET DISPLAYS
00027      FCBC  REDIS EQU  $FCBC
00028 0026 8D 17  BSR  PRINT
00029 0028 8D 15  BSR  PRINT
00030 002A 8D 13  BSR  PRINT  PRINT HOURS,MINUTES,SECONDS
00031 002C 20 DB  BRA  TIMPAS  DO IT ALL AGAIN
00032      ** INCR - INCREMENT SUBROUTINE
00033 002E A6 00  INCR  LDA  A  0,X  DATA WORD INTO A
00034 0030 89 00  ADC  A  #0  INCREMENT IF NECESSARY
00035 0032 19  DAA  FIX TO DECIMAL
00036 0033 11  CBA  TIME TO CLEAR?
00037 0034 25 01  BCS  INC1  NO
00038 0036 4F  CLR  A
00039 0037 A7 00  INC1  STA  A  0,X
00040 0039 08  INX
00041 003A 07  TPA
00042 003B 88 01  EOR  A  #1  COMPLEMENT CARRY BIT
00043 003D 06  TAP
00044 003E 39  RTS
00045      ** PRINT - PRINT HEX BYTES
00046 003F 09  PRINT DEX  POINT X AT BYTE
00047 0040 96 02  LDA  A  $02  WHAT'S IN HOURS?
00048 0042 26 03  BNE  CONTIN  IF NOT ZERO
00049 0044 7C 0002  INC  HOURS  MAKE IT ONE
00050 0047 A6 00  CONTIN LDA  A  0,X
00051 0049 7E FE20  JMP  OUTBYT
00052      FE20  OUTBYT EQU  $FE20  MONITOR ROUTINE
00053 00F7  ORG  $00F7
00054 00F7 3B  RTI
00055      END

```

Figure 10-62  
 Clock program with audible tick-tock.

## Procedure (continued)

10. Switch the Trainer power off. Then remove the wire interconnecting LINE and  $\overline{\text{IRQ}}$ . Switch the Trainer power on.
11. Write a program that will output the proper data to produce an audio tone from the speaker circuit. This program must: Initialize the PIA, alternately store 1's and 0's to the speaker in order to produce a tone, and provide a delay loop between each storage, to determine the frequency of the tone.
12. Execute the program.

## Discussion

Figure 10-63 shows a program similar to the one you wrote. Notice that only two instructions were required to initialize the PIA. This is possible since only the B side will be used to output data.

Remember from the previous program, it is only necessary to change the  $D_0$  bit of the output data, since that is the only bit connected to the speaker circuit. Therefore, you can start with a random number in the A accumulator (line 00007) and store the number to the PIA. After a short delay (lines 00008 thru 00010) the A accumulator is incremented (changing the  $D_0$  bit logic level) and again stored to the PIA. This incrementing and storing of accumulator A can continue indefinitely since the only data of interest is the  $D_0$  bit.

```
00001          NAM      TONETEST REV. 0.2
00002          OPT      NOP
00003          *INITIALIZE PIA
00004 0000 CE FF04      LDX      #$FF04
00005 0003 FF 8002      STX      $8002
00006          *PRODUCE TONE
00007 0006 B7 8002 ALTERN STA A  $8002      OUTPUT BIT
00008 0009 C6 55      LDA B  #$55      DETERMINES FREQUENCY
00009 000B 5A          TONE  DEC B
00010 000C 26 FD      BNE      TONE
00011 000E 4C          INC A          COMP. BIT
00012 000F 20 F5      BRA      ALTERN
00013          END
```

Figure 10-63

Program to output a tone through the speaker.

## Procedure (continued)

13. Enter the program listed in Figure 10-64. After you enter  $F1_{16}$  at address  $0024_{16}$ , go to address  $0101_{16}$  and enter the remaining data bits. Notice that the program covers two pages. This listing second page has been condensed to show only the assembled program line numbers, addresses, and data. The data in addresses 000D and 000E controls the time of each note. The number in parentheses is for the **fast clock**. The number in addresses 0101 through 01BF determine the pitch, or frequency, of each note. Once more, the numbers in parentheses are for the **fast clock**.
14. Execute the program beginning at address  $0000_{16}$ . Notice that after the program completes the song, there is a pause (of equal duration to the song) before the song repeats.

00001				0107 53 (9D)
00002 0000				0108 42 (7C)
00003 0000 7F 8003	CLR			0109 53 (9D)
00004 0003 7C 8002	INC		00023 010A 42 (7C)	010B 53 (9D)
00005 0006 73 8003	COM			010C 42 (7C)
00006 0009 8E 0100	LDS			010D 37 (69)
00007 000C CE 05FF (0CFF)	LDX			010E 42 (7C)
00008 000F 33	PUL B			010F 37 (69)
00009 0010 5D	TST B			0110 42 (7C)
00010 0011 27 ED	BEQ			0111 37 (69)
00011 0013 F7 0100	STA B			0112 42 (7C)
00012 0016 4C	INC A		00024 0113 37 (69)	
00013 0017 F6 0100	LDA B			0114 42 (7C)
00014 001A 09	DEX		00025 0115 22 (41)	
00015 001B 27 EF	BEQ			0116 2B (52)
00016 001D 5A	DEC B			0117 22 (41)
00017 001E 26 FA	BNE			0118 2B (52)
00018 0020 B7 8002	STA A			0119 20 (3D)
00019 0023 20 F1	BRA			011A 29 (4D)
00020 0100	ORG			011B 20 (3D)
00021 0100 0001				011C 29 (4D)
00022 0101 53 (9D)			00026 011D 20 (3D)	
0102 42 (7C)				011E 29 (4D)
0103 53 (9D)				011F 20 (3D)
0104 42 (7C)				0120 29 (4D)
0105 53 (9D)				0121 20 (3D)
0106 42 (7C)				

Figure 10-64  
Music program (Part 1 of 2).



0122 29 (4D)	00031 014A 4A (8C)	0172 4A (8C)	019A 46 (84)
0123 20 (3D)	014B 3E (75)	0173 37 (69)	019B 3A (6E)
0124 29 (4D)	014C 4A (8C)	0174 4A (8C)	019C 46 (84)
0125 20 (3D)	014D 3E (75)	00036 0175 31 (5C)	019D 3E (75)
00027 0126 29 (4D)	014E 4A (8C)	0176 3E (75)	019E 4A (8C)
0127 20 (3D)	014F 3E (75)	0177 31 (5C)	019F 3E (75)
0128 29 (4D)	0150 4A (8C)	0178 3E (75)	01A0 4A (8C)
0129 20 (3D)	0151 3E (75)	0179 2B (52)	01A1 3E (75)
012A 29 (4D)	0152 4A (8C)	017A 37 (69)	00041 01A2 4A (8C)
012B 20 (3D)	00032 0153 3E (75)	017B 2B (52)	01A3 3E (75)
012C 29 (4D)	0154 4A (8C)	017C 37 (69)	01A4 4A (8C)
012D 20 (3D)	0155 3E (75)	017D 2B (52)	01A5 24 (45)
012E 29 (4D)	0156 4A (8C)	00037 017E 37 (69)	01A6 3E (75)
00028 012F 20 (3D)	0157 3E (75)	017F 2B (52)	01A7 24 (45)
0130 29 (4D)	0158 4A (8C)	0180 37 (69)	01A8 3E (75)
0131 2B (52)	0159 3E (75)	0181 2B (52)	01A9 29 (4D)
0132 37 (69)	015A 4A (8C)	0182 37 (69)	01AA 42 (7C)
0133 2B (52)	015B 3E (75)	0183 2B (52)	00042 01AB 29 (4D)
0134 37 (69)	00033 015C 4A (8C)	0184 37 (69)	01AC 42 (7C)
0135 24 (45)	015D 3E (75)	0185 2B (52)	01AD 29 (4D)
0136 2B (52)	015E 4A (8C)	0186 37 (69)	01AE 42 (7C)
0137 24 (45)	015F 3E (75)	00038 0187 2B (52)	01AF 29 (4D)
00029 0138 2B (52)	0160 4A (8C)	0188 37 (69)	01B0 42 (7C)
0139 29 (45)	0161 3E (75)	0189 2B (52)	01B1 29 (4D)
013A 37 (69)	0162 5B (A7)	018A 37 (69)	01B2 42 (7C)
013B 29 (45)	0163 3E (75)	018B 2B (52)	01B3 29 (4D)
013C 37 (69)	0164 5B (A7)	018C 37 (69)	00043 01B4 42 (7C)
013D 42 (7C)	00034 0165 3E (75)	018D 2B (52)	01B5 29 (4D)
013E 37 (69)	0166 5B (A7)	018E 37 (69)	01B6 42 (7C)
013F 42 (7C)	0167 3E (75)	018F 2B (52)	01B7 29 (4D)
0140 37 (69)	0168 5B (A7)	00039 0190 37 (69)	01B8 42 (7C)
00030 0141 42 (7C)	0169 3E (75)	0191 31 (5C)	01B9 2B (52)
0142 37 (69)	016A 5B (A7)	0192 3E (75)	01BA 31 (5C)
0143 42 (7C)	016B 3E (75)	0193 31 (5C)	01BA 37 (69)
0144 37 (69)	00035 016C 5B (A7)	0194 3E (75)	01BC 3E (75)
0145 3A (6E)	016D 37 (69)	0195 37 (69)	00044 01BD 42 (7C)
0146 46 (84)	016E 4A (8C)	0196 42 (7C)	01BE 4A (8C)
0147 3A (6E)	016F 37 (69)	0197 37 (69)	01BF 00 (00)
0148 46 (84)	0170 4A (8C)	0198 42 (7C)	00045 END
0149 3E (75)	0171 37 (69)	00040 0199 3A (6E)	

Figure 10-64  
 Music program (Part 2 of 2).

## Discussion

The program you entered occupies memory locations 0000 through 0024<sub>16</sub>. The remaining data represents the notes in the music. It was structured in this manner so that you could experiment with different songs. Figure 10-65 illustrates the various notes the program can produce, on the outline of an organ keyboard. Each note is listed with its actual fundamental frequency below the note letter. The number below the frequency is the hex number that will produce that approximate frequency. The number in parentheses is the one to use if your Trainer has been modified and has a **fast clock**. The notes your Trainer will produce depends on the MPU clock frequency. Even with the crystal-controlled clock in the modified Trainer, it is not possible to reproduce the exact frequency of the notes.

Although the music program is basically simple, there are a few unique features that should be examined. The first instruction clears control register B of the PIA. Naturally, this occurs prior to program execution. However, it will be necessary to modify the contents of data direction register B prior to each program cycle. Thus, bit two in the control register is cleared.

The second instruction turns bit PB0 on or off for each program cycle. Incrementing the data direction register will be of more value in the next section of this experiment.

Instruction four (LDS) tells the MPU that the data stored at 0101 thru 01BF<sub>16</sub> now resides in the stack. However, the pointer contains address 0100<sub>16</sub>. This is necessary, since each "pull" instruction adds "1" to the pointer prior to execution.

The last note in the stack is 00<sub>16</sub>. This is used to indicate "end of music." Since a pull instruction does not affect any of the MPU condition codes, it is necessary to test for zero with instruction seven (TST B).

F#	G#	A#	C#	D#	F#	G#	A#	C#	D#	F#	G#	A#	C#
185.0	207.7	233.1	277.2	311.1	370.0	415.3	466.2	554.4	622.3	740.0	830.6	932.3	1108.7
76	69	5D	4E	46	3A	34	2E	27	22	1D	19	17	13
(DF)	(C6)	(B1)	(94)	(84)	(6E)	(62)	(57)	(49)	(41)	(36)	(30)	(2B)	(24)
F	G	A	B	C	D	E	F	G	A	B	C	D	E
174.6	196.0	220.0	246.9	261.6	293.7	329.6	349.2	392.0	440.0	493.9	523.3	587.3	659.3
7D	6F	63	58	53	4A	42	3E	37	31	2B	29	24	20
(ED)	(D2)	(BB)	(A7)	(9D)	(8C)	(7C)	(75)	(69)	(5C)	(52)	(4D)	(45)	(3D)
F	G	A	B	C	D	E	F	G	A	B	C	D	E
698.5	784.0	880.0	987.8	1046.5	1174.7								
1E	1B	18	15	14	12								
(39)	(33)	(2D)	(28)	(26)	(21)								

Figure 10-65

Music notes reproduced by the program in Figure 10-64.

The remaining program steps contain two timing loops. The first, starting at line 00007 sets the music tempo. The second, starting at line 00008 produces the notes.

NOTE: If you wish to listen to your ROM, enter  $FC_{16}$  at  $000A_{16}$ , and  $01_{16}$  at  $0010_{16}$ . It is necessary to remove the TST B instruction, since ROM contains a number of  $00_{16}$  data bytes. Now, the program will continue until you press RESET.

## Procedure (continued)

15. If you modified the music program (addresses 0000 through  $0024_{16}$ ), refer to Figure 10-64 and reenter the program. Its not necessary to reenter the same music notes if you have not modified them.
16. Refer to Figure 10-66 and modify your speaker circuit. Notice that a resistor is placed between gate IC3B and the speaker. Also, an additional gate (from IC3) and resistor interface pin 11 of the PIA with the speaker.
17. Execute the music program. This time, the music plays three times before there is a pause. The first repeat is an octave lower, and the second repeat simultaneously plays the original and octave lower music.

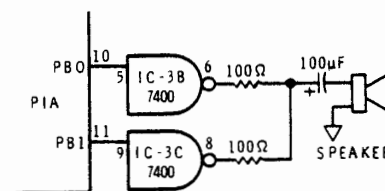


Figure 10-66  
Modification to the speaker circuit.

## Discussion

The gate connected to pin 11 of the PIA (data bit PB1) allows an additional output interface to the speaker. The two resistors reduce circuit loading so the outputs of the two gates can be combined at the coupling capacitor. However, the resistors also reduce the signal level and as a result, speaker volume.

The music program is unchanged from the previous section. However, you are now using two additional features in the program that previously were not required or apparent. The first concerns the PIA. Each time the program repeats, the data direction register bits are incremented. This meant the PB0 bit cycled the music on and off. Now that two output pins are wired to the circuit, a new pattern develops. First, pin PB0 is active. Then pin PB1 is active. Next, both pins are active. Finally both pins are inactive. Thereafter, the cycle repeats.

That cyclic pattern accounts for two (apparent) channels of music. But, why does one channel sound like it is one octave lower in frequency?

At the end of each "note" timing loop, the A accumulator is stored, and then incremented. Thus, the speaker is driven by a cyclic logic level transition of the  $D_0$  data bit. However, every two level transitions in the  $D_0$  bit causes a single level transition in the  $D_1$  bit. Figure 10-67 illustrates the process. Since bit  $D_0$  is coupled to PIA bit PB0, and bit  $D_1$  is coupled to PIA bit PB1, you effectively have identical music material generated at two different octave levels.

If you have a stereo sound system, you can connect the music output of each gate (through a 100 ohm resistor) to the AUX input of your amplifier. The sound reproduction will be better than that produced by your Trainer.

This completes this experiment. Leave the circuit intact for the next experiment.

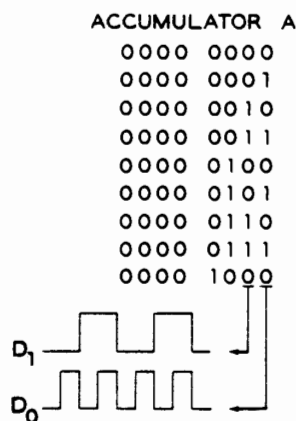


Figure 10-67

Music material coupled to the PIA.

## Experiment 8

### **KEY MATRIX AND PARALLEL-TO-SERIAL CONVERSION**

#### **OBJECTIVES:**

- Demonstrate a method for using any combination of PIA I/O ports as inputs or outputs.*
- Show how a matrix-type keyboard decoder system works, and how it can be constructed.*
- Demonstrate parallel-to-serial conversion using the PIA.*
- Demonstrate a method for converting a hex digit to ASCII.*
- Show how a "one-shot" monostable works, using software.*
- Show how to add the parity bit to a serial word, using software.*

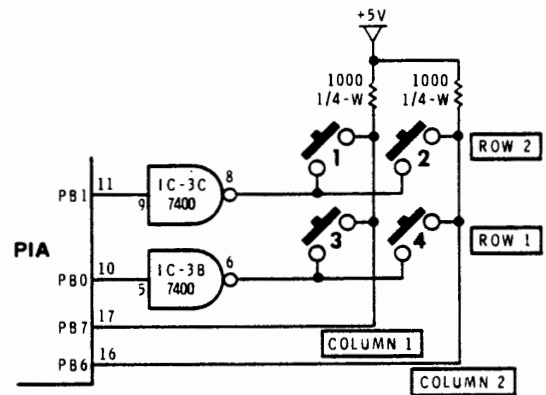
#### **Introduction**

You are quite familiar with the PIA by now. In this experiment, you will demonstrate its versatility as an I/O device. In addition to using one peripheral port as both an input and output bus, you will see how a parallel data transfer device can be used to communicate in "serial." Since a great amount of serial data uses ASCII, this experiment will use the ASCII format.

#### **Material Required**

- 1 ET-3400 Microprocessor Trainer with PIA circuit wired to the large connector block
- 2 1000 ohm, 1/4-watt, 10% resistors
- 1 Pushbutton switch #1
- 1 Pushbutton switch #2
- 1 Pushbutton switch #3
- 1 Pushbutton switch #4

Figure 10-68  
Matrix switch circuit.



## Procedure

1. In this part of the experiment, you will see how the PIA interfaces with a switch matrix. Switch the Trainer power off. Then remove the 100  $\mu$ F capacitor, two 100 ohm resistors, speaker and the wires associated with those parts.

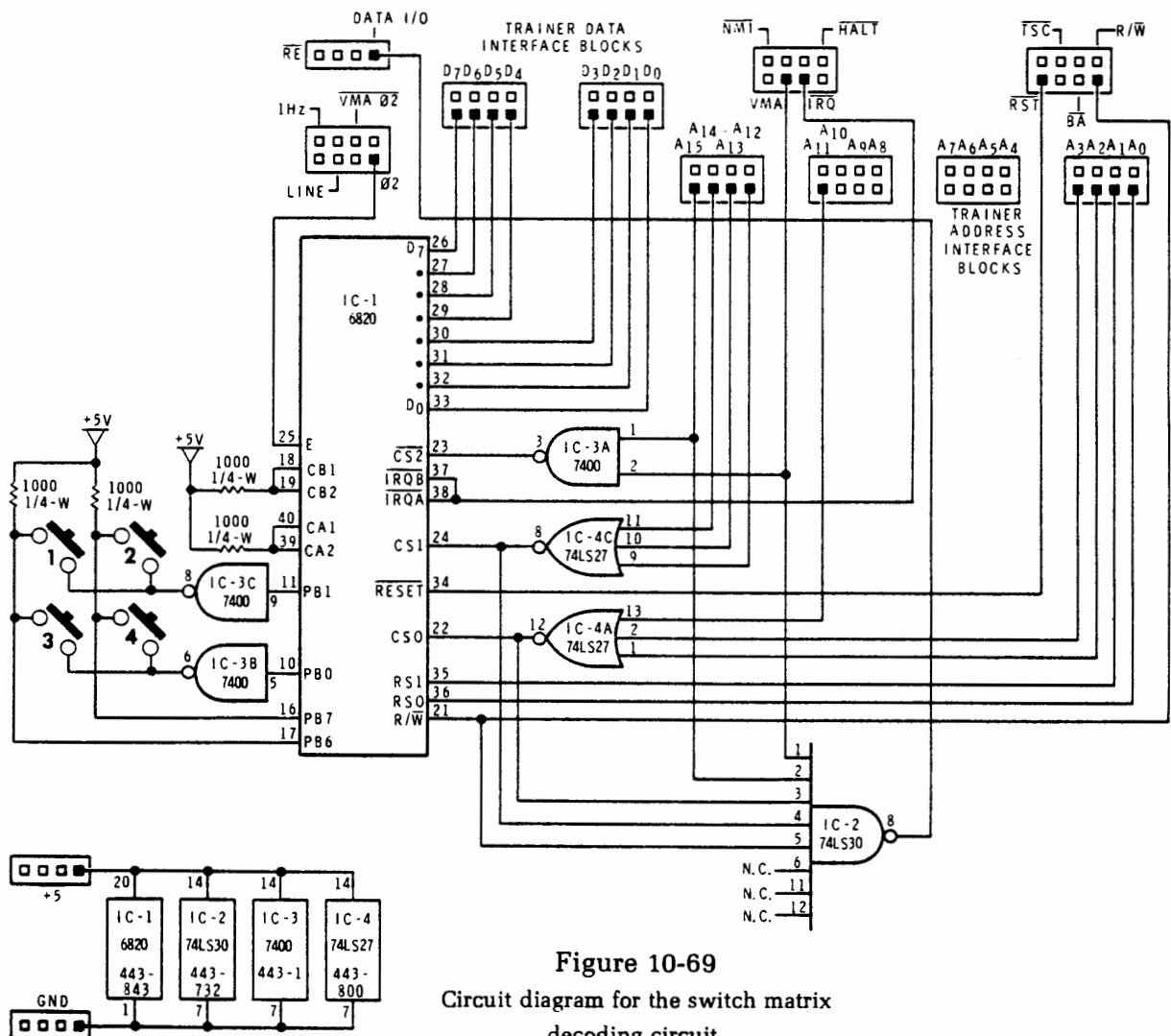


Figure 10-69  
Circuit diagram for the switch matrix  
decoding circuit.

2. Refer to Figure 10-68 and add the circuit shown to the PIA circuit. Figure 10-69 shows the complete circuit wired to your Trainer.
3. Refer to Figure 10-70 and enter the program listed. Do not attempt to enter data at address 003E<sub>16</sub>. This will serve as a temporary storage register.
4. Execute the program. The display will go blank, except for the decimal point in digit H.
5. Randomly press the matrix circuit pushbuttons, individually and in combination, and observe display H.

```

00001          NAM      KEYMATRI REV 0.4
00002          OPT      NOP
00003      FE28      OUTHEX EQU      $FE28
00004      FCBC      REDIS  EQU      $FCBC
00005      FE52      OUTST1 EQU      $FE52
00006          *INITIALIZE PIA
00007 0000 CE 0F04      LDX      #$0F04
00008 0003 FF 8002      STX      $8002
00009          *SET DISPLAY LOCATION
00010 0006 BD FCBC NEWKEY JSR      REDIS      SET DISPLAY LOCATION
00011          *REFRESH WAIT
00012 0009 86 FF          LDA A      $FF      *
00013 000B 4A          WAIT  DEC A      *WAIT
00014 000C 26 FD          BNE      WAIT      *
00015          *NUMBER OF KEYS
00016 000E 86 04          LDA A      $04      TOTAL KEYS
00017 0010 B7 003E      STA A      KEYNUM    TO UPDATE
00018          *ROW SEARCH
00019 0013 86 01          LDA A      $01      SELECT ROW ONE
00020 0015 B7 8002      STA A      $8002      OUTPUT TEST BIT
00021 0018 F6 8002 NXTROW LDA B      $8002      GET CLOSURE
00022 001B 2A 10          BPL      COL1      WAS IT COL1?
00023 001D 59          ROL B
00024 001E 2A 10          BPL      COL2      WAS IT COL2?
00025 0020 7A 003E      DEC      KEYNUM
00026 0023 7A 003E      DEC      KEYNUM
00027 0026 27 10          BEQ      OUT      NO KEY INDICATION
00028 0028 78 8002      ASL      $8002      SHFT ROW BIT
00029 002B 20 EB          BRA      NXTROW    TEST NEXT ROW
00030          *COLUMN IDENTIFICATION
00031 002D 7A 003E COL1  DEC      KEYNUM
00032 0030 B6 003E COL2  LDA A      KEYNUM    GET KEY NUMBER
00033 0033 BD FE28      JSR      OUTHEX      MONITOR ROUTINE
00034 0036 20 CE          BRA      NEWKEY
00035          *BLANK DISPLAY
00036 0038 BD FE52 OUT  JSR      OUTST1      MONITOR ROUTINE
00037 003B 80          FCB      $80          OUTPUT D.P. ONLY
00038 003C 20 C8          BRA      NEWKEY
00039 003E 0001 KEYNUM RMB      1      NUMBER TO BE DISPLAYED
00040          END

```

Figure 10-70  
Program for decoding key matrix.

## Discussion

Refer to Figure 10-68. Notice that each pushbutton switch occupies a unique row and column electrically. Referring to the switch contacts, each "column" line is held at a logic 1 through a 1000 ohm resistor. Each "row" line is held at a logic 1 by the output of a NAND gate. In this circuit, the gates serve as inverter/drivers.

Assume that you close switch 2. In searching for the closed switch, the program first places a logic 0 on row 1 and then examines column 1 and column 2. Since both columns are still at a logic 1, row 1 is returned to logic 1, and row 2 is pulled to a logic 0. Again columns 1 and 2 are examined. This time, column 2 is found to be low, indicating that switch 2 is closed.

Refer to the program in Figure 10-70. Data direction register B is loaded with a bit pattern ( $0F_{16}$ ) that sets pins PB0 and PB1 as outputs, and pins PB6 and PB7 as inputs. Although the other pins are set, they are not used in this experiment. Bit pattern  $04_{16}$  then sets bit 2 of the control register high for access to the peripheral B interface.

A jump to monitor routine REDIS stores the address of display H in a temporary location in memory. This address will be used by other monitor routines to output data to display H.

The next three instructions provide a short time delay to help prevent character "ghosting." Some ghosting may be noticed in subdued light, due to the "rewriting" techniques used in this program.

Since four pushbutton switches (keys) are used in this experiment, decimal 4 is stored at temporary register  $0041_{16}$ . This number or a decremented value of this number will be displayed when a switch closure is recognized.

The row search begins by storing  $01_{16}$  to the PIA. This pulls the output of IC3B low (row 1). Then the PIA B side bus data is loaded into the B accumulator, and bit  $D_7$  is tested for a 0. Assuming switch 2 was pressed, bit  $D_7$  will test as a 1. The B accumulator is rotated left so that bit  $D_6$  can be tested at the  $D_7$  position. Since it also indicates 1, switches 3 and 4 have tested open.

Key number 4 in the temporary register ( $0041_{16}$ ) is decremented twice prior to testing switches 1 and 2. Data stored at the PIA is shifted left. This pulls row 2 low.



The row search begins again at line 00021. When column 2 is tested, it is discovered that switch 2 is closed. The program branches to address 0033<sub>16</sub>, and the A accumulator is loaded with the key number (2) from the temporary register. The monitor routine, OUTHEX, writes the number 2 into display H, and branches back to address 0006<sub>16</sub> where the program begins again.

If all of the keys test open during row search, the program will branch to OUT, where a monitor routine lights the decimal point in display H and blanks all of the other displays. Then, the program branches back to address 0006<sub>16</sub> and begins again.

You may have noticed that as you press more than one switch, the first switch tested will have priority. This occurs in a 3, 4, 1, 2 sequence.

Up to 16 switches can be tested with this program. By using both the A and B sides of the PIA, up to 64 switches can be tested. This can represent a big savings in peripheral interface logic.

## Procedure (continued)

6. Switch the Trainer power off. Then remove the four switches, their two 1000 ohm resistors, and their associated wires. This includes the wires at pins 10 and 11 of the PIA.
7. Add the circuit shown in Figure 10-71 to the PIA circuit wired to your Trainer.

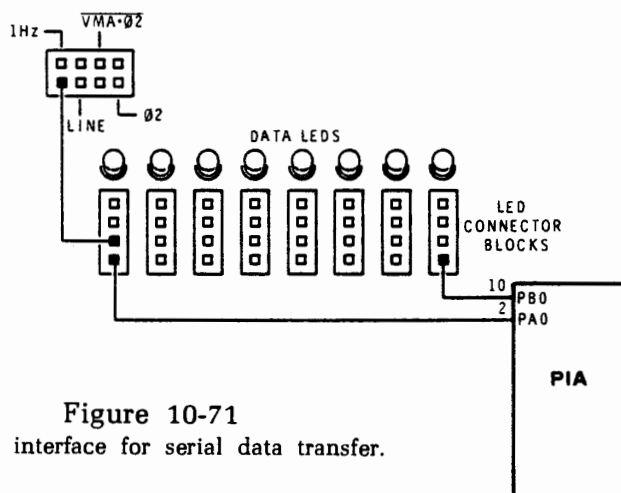


Figure 10-71  
Display interface for serial data transfer.

8. Switch Trainer power on. Then enter the program listed in Figure 10-72. Do not attempt to enter data at address 0048<sub>16</sub>. This address will serve as a temporary storage register.

When this program is executed, data LED7 will flash on and off at approximately a 1 Hz rate. The 1 Hz signal will be used as a program timing signal through PIA pin PA0. The flashing LED will serve as a visual timing reference.

```

00001          NAM      SERIAL01 REV 0.6
00002          OPT      NOP
00003      8000      PIAIN EQU      $8000
00004      FDF4      INCH  EQU      $FDF4
00005      8002      PIAOUT EQU     $8002
00006 0000 CE FE04      LDX      $$FE04
00007 0003 FF 8000      STX      PIAIN
00008 0006 CE FF04      LDX      $$FF04
00009 0009 FF 8002      STX      PIAOUT
00010 000C 7F 0048      CLR      TEMP
00011 000F 73 8002      COM      PIAOUT
00012          *GET HEX CHARACTER
00013 0012 BD FDF4 NXTCHR JSR      INCH
00014 0015 01          NOP          * 8D      HOLD
00015 0016 01          NOP          * 39      FOR
00016 0017 01          NOP          * 8D      PROGRAM
00017 0018 01          NOP          * 47      MODIFICATION
00018          *COMMENCE WITH START BIT
00019 0019 7F 8002 RESET CLR      PIAOUT  RESETS ALL OUT BITS
00020 001C 8D 1B      BSR      DELAY
00021          *OUTPUT THE CHARACTER
00022 001E B7 8002      STA A      PIAOUT  LSB IS STORED OUT
00023 0021 8D 16      BSR      DELAY
00024 0023 86 07      LDA A      #07      NO. OF TIMES SHIFTED
00025 0025 76 8002 WORD ROR      PIAOUT  NEXT MSB IS STORED OUT
00026 0028 8D 0F      BSR      DELAY
00027 002A 4A          DEC A          *CHECK FOR NUMBER OF
00028 002B 26 F8      BNE      WORD  *BITS SHIFTED
00029          *OUTPUT 2 STOP BITS
00030 002D 86 01      LDA A      $$01      SET LSB
00031 002F B7 8002      STA A      PIAOUT  STORE IT TO OUTPUT
00032 0032 8D 05      WAIT BSR      DELAY
00033 0034 4A          DEC A          *WAIT FOR TWO
00034 0035 26 FB      BNE      WAIT  *BIT TIMES
00035 0037 20 D9      BRA      NXTCHR  DONE! GET NEXT CHAR.
00036          *DELAY SUBROUTINE
00037 0039 F6 8000 DELAY LDA B      PIAIN  SAMPLE INPUT LOGIC LVL.
00038 003C 50          NEG B          INPUT NOW FF OR 00
00039 003D F1 0048      CMP B      TEMP  IS IT LIKE TEMP?
00040 0040 27 F7      BEQ      DELAY  IF SO, GET ANOTHER SAMPLE
00041 0042 73 0048      COM      TEMP  IF NOT, COMP TEMP
00042 0045 2B F2      BMI      DELAY  IF TEMP = FF, STAY IN LOOP
00043 0047 39          RTS
00044 0048 0001      TEMP RMB      1
00045          END

```

Figure 10-72

Program to input serial data.

9. Execute the program. Data LED0 will light to indicate the program is running.

## Discussion

Until now, you have observed data words being displayed in a parallel manner only. That is, all of the bits contained in the data word or byte were displayed simultaneously. You will now display a data word *serially*. In the serial mode, data bits are displayed one after the other. For this experiment, the bits-per-second or baud rate will be very slow so that you will be able to recognize each data bit. Baud is defined as one bit per second.

When you return to the experiment, you will use the Trainer keyboard to enter a hex number. The number will be converted to its binary form and transferred to data LED 0 one bit at a time. However, what you will see is not a 4-bit word, but rather an 11-bit word.

When serial data is transferred, it must fulfill certain format conventions. These include a start bit, to indicate the beginning of a word; the information being transferred; and a stop bit, to indicate the end of a word. Depending on the instrument sending data, and the baud rate, the data word can have one start bit, six, seven, or eight data bits, zero or one parity bit (often considered one of the data bits) and one or two stop bits.

The word format used in this experiment uses one start bit, eight data bits, and two stop bits. Figure 10-73 illustrates the serial word for hex 5. For added clarity, the timing signal for data LED7 is included. Notice that the actual data is transferred, beginning with the LSB. Also, only the first four data bits (plus the start bit) are of interest, since you are only transferring a single hex digit. Later in the experiment, the remaining four data bits will be used.

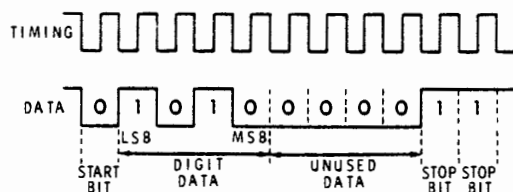


Figure 10-73

Serial data word format for the number 5<sub>16</sub>.

## Procedure (continued)

10. All timing is referenced to data LED7. Serial data is transferred through data LED0. When you are instructed to press a Trainer key, momentarily press the key while LED7 is **off**. It will take a little practice to be able to identify the data being transferred. Refer to Figure 10-73; it shows the data word you will observe when you press the 5 key. Press the 5 key and observe data LED's 0 and 7. Do it a number of times so that you can recognize each bit in the serial word.
11. Press a number of different Trainer keys and observe each serial word. You may find it helpful to illustrate each word, so you know what to expect.

## Discussion

Data direction register A is set so all of the pins of peripheral bus A are outputs, except for pin PA0. Thus, it will not be necessary to electrically tie the unused pins to a logic 1. Data direction register B is set so all of the pins of peripheral bus B are outputs. Then, temporary register 0048<sub>16</sub> is cleared.

Since output register B contains logic 0's from PIA reset, the contents of the register are complemented. This turns data LED0 on, its waiting condition. Everything is now prepared, and the main program can begin.

The program immediately jumps to monitor routine INCH and waits for an input from the Trainer keyboard. When a key is pressed, PIA pin PB0 is cleared, generating a start bit. The delay used to set the bit time will be described at the end of the program.

After the start bit delay, the key number, now residing in accumulator A, is stored in output register B. Data LED0 immediately displays the LSB of the number. Then accumulator A is loaded with the number representing the number of times output register B must be rotated right in order to display each data bit in data LED0. The data in output register B is rotated through pin PB0 with a time delay after each rotate.

Once all eight data bits have been stored, 01<sub>16</sub> is stored to output register B from the A accumulator. This forces LED0 on, indicating the first stop bit. After two time delays, for the two stop bits, the program branches back to monitor routine INCH and waits for a new key closure.

The delay subroutine uses the 1 Hz clock for timing. This is why the serial data transfer coincides with the lighting of data LED7.

At the beginning of the delay routine, PIA peripheral interface A is examined. Pin PA0 should be logic 0 if you pressed the Trainer key while LED7 was off (logic 0). Therefore,  $00_{16}$  is stored in the B accumulator. ( $01_{16}$  would be stored if LED7 was on.)

The data in the B accumulator is negated and then compared with the temporary register ( $0048_{16}$ ). Since both registers are equal, the program loops back and examines pin PA0 again. This cycle continues until pin PA0 goes to a logic 1. Accumulator B is loaded with  $01_{16}$ . The data is negated to produce  $FF_{16}$ .

Since the B accumulator and temporary register are no longer equal, the temporary register is complemented. This changes its contents to  $FF_{16}$ , which represents a negative number to the MPU. Because the temporary register contains a negative number, the program branches back to the beginning of the delay routine.

Pin PA0 is again repeatedly examined for a logic level change from 1 to 0. When this level transition occurs, the program returns the temporary register to its original  $00_{16}$  condition, and then returns the program counter to the main program. Thus, you have effectively generated a software one-shot monostable with a time period determined by the 1 Hz signal.

## Procedure (continued)

12. Enter the program listed in Figure 10-74. Notice that the program begins at address  $0050_{16}$ .

00001		NAM	ASCICONV REV 0.1
00002		OPT	NOF
00003	0050	ORG	\$50
00004		*CONVERTS HEX TO ASCII	
00005	0050 8A 30	ORA A	##30 ASC NO. START WITH 3
00006	0052 81 39	CMF A	##39 IS IT A NUMBER?
00007	0054 23 04	BLS	DONE IF SO, DONE
00008	0056 80 09	SUB A	##09 *LETTER START AT 1
00009	0058 8B 10	ADD A	##10 *AND BEGIN WITH 4
00010	005A 39	DONE	RTS
00011		END	

Figure 10-74

Program to modify serial program for  
ASCII word format.

13. Now return to address  $0015_{16}$  and enter  $8D_{16}$ . Then enter  $39_{16}$  at address  $0016_{16}$ .
14. Execute the program beginning at address  $0000_{16}$ . This is the beginning of the serial output program.
15. A hex to ASCII conversion subroutine has been added to the serial output program. Refer to Figure 10-75. Notice that the ASCII representation for hex 5 is  $35_{16}$ . Press the Trainer's 5 key and watch data LED0. Be sure to press the key while data LED7 is off. The serial data format remains unchanged, with one start bit, eight data bits, and two stop bits.
16. Press a number of Trainer keys, and observe the serial transfer for each key.

COLUMN		0	1	2	3	4	5	6	7
ROW	BITS 765 4321	000	001	010	011	100	101	110	111
0	0000	NUL	DLE	SP	0	@	P	\	p
1	0001	SOH	DC1	!	1	A	Q	a	q
2	0010	STX	DC2	"	2	B	R	b	r
3	0011	ETX	DC3	#	3	C	S	c	s
4	0100	EOT	DC4	\$	4	D	T	d	t
5	0101	ENQ	NAK	%	5	E	U	e	u
6	0110	ACK	SYN	&	6	F	V	f	v
7	0111	BEL	ETB	'	7	G	W	g	w
8	1000	BS	CAN	(	8	H	X	h	x
9	1001	HT	EM	)	9	I	Y	i	y
10	1010	LF	SUB	*	:	J	Z	j	z
11	1011	VT	ESC	-	;	K	[	k	{
12	1100	FF	FS	.	<	L	\	l	
13	1101	CR	GS	_	=	M	]	m	~
14	1110	SO	RS	`	>	N	^	n	~
15	1111	SI	US	/	?	O	_	o	DEL

Figure 10-75

Table of 7-bit American Standard Code  
for Information Interchange.

## Discussion

Remember that the Trainer outputs the hex value in binary when a number is pressed. Therefore, when the 5 key is pressed,  $05_{16}$  will be loaded into the A accumulator, in the serial output program. Then the program will branch to the ASCII conversion routine. This is caused by the branch instruction in address  $0015_{16}$  and the **relative** address for the branch in address  $0016_{16}$ .

The first instruction in the conversion routine OR's  $05_{16}$  with  $30_{16}$  to produce  $35_{16}$ . The next instruction compares  $35_{16}$  with  $39_{16}$ . If the first value is smaller (which it is) or equal to  $39_{16}$ , a valid ASCII number is present in the A accumulator, and the program counter is sent back to the main program.

If a valid number is not present,  $09_{16}$  is subtracted from the value in the A accumulator. Assume that the C key was pressed. Then the number stored in the A accumulator is  $0C_{16}$ . When OR'ed with  $30_{16}$ , it equals  $3C_{16}$ . Remember that a compare instruction does not alter the contents of the accumulator. Therefore, when  $09_{16}$  is subtracted from the contents of the A accumulator, the result is  $3C_{16} - 09_{16} = 33_{16}$ . Finally,  $10_{16}$  is added to the contents of the A accumulator, resulting in the value  $43_{16}$ . Notice in Figure 10-75 that  $43_{16}$  equals C in ASCII code. Again the program counter returns to the main program.

## Procedure (continued)

17. Enter the program listed in Figure 10-76. Notice that this program begins at address  $0060_{16}$ . Stop after you enter  $39_{16}$  at address  $0075_{16}$ . Address  $0076_{16}$  is used as a temporary register.

00001		NAM	ADDPARIT	REV 0.1	
00002		OPT	NOF		
00003	0060	ORG	\$60		
00004			*ADD PARITY BIT		
00005	0060 7F 0076	PARITY CLR	PARIT1	START FRESH	
00006	0063 C6 09	LDA B	#\$09	TIMES TO SHIFT	
00007	0065 49	BITCNT ROL A		SHIFT ONCE	
00008	0066 24 03	BCC	NOINCR	SKIP INCR.	
00009	0068 7C 0076	INC	PARIT1	COUNT LOGIC 1 BITS	
00010	006B 5A	NOINCR DEC B		COUNT OFF TOTAL BITS	
00011	006C 26 F7	BNE	BITCNT	ALL CHECKED YET?	
00012	006E 76 0076	ROR	PARIT1	CHECK LSB OF PARIT1	
00013	0071 24 02	BCC	FINIS	WAS IT ODD?	
00014	0073 8A 80	ORA A	#\$80	IF SO, SET PARITY BIT	
00015	0075 39	FINIS RTS			
00016	0076 0001	PARIT1 RMB	1		
00017		END			

Figure 10-76

Subroutine to add parity bit to serial output program.

18. Now return to address  $0017_{16}$  and enter  $8D_{16}$ . Then enter  $47_{16}$  at address  $0018_{16}$ .
19. Execute the program beginning at address  $0000_{16}$ . This is the beginning of the serial output program.

## Discussion

Remember from Unit 1 that a parity bit is added to a serial word as a data transfer check. It indicates whether the sum of logic 1's in the data portion of the word are odd or even. Thus, if you desire an **even** parity check, the sum of the parity bit and all of the other data bits must equal an even number. Odd parity requires that all of the data bits plus the parity bit equal an odd number.

For example, the ASCII code for the number 5 is  $35_{16}$ . Since  $35_{16}$  equals  $0011\ 0101_2$ , the parity bit would be 0 for even parity and 1 for odd parity.

The parity bit occupies the eighth data bit position in the 8-bit data word. The first seven data bits contain the ASCII code.

## Procedure (continued)

19. Determine the serial data word for hex 5. Then press the 5 key and observe LED0. Notice that the parity bit is 0 since the parity routine is configured for even parity.
20. Press a number of Trainer keys, and observe the serial transfer for each key.
21. Change the program data address  $0071_{16}$  to  $25_{16}$ . then press a number of Trainer keys, and observe the serial transfer for each key. The parity routine is now configured for odd parity.

## Discussion

After the hex number is converted to ASCII code, the program branches from address  $0017_{16}$  to the "even" parity routine. This routine determines if a 1 or 0 will be placed in the eighth data bit to provide an even parity indication.

To begin, the temporary register at address  $0076_{16}$  is cleared. This register will store the count of the number of logic 1 bits found in the data word. Accumulator B is loaded with  $09_{16}$ , which will be decremented to monitor the bit count routine.



Accumulator A is rotated left and the carry bit is checked. If the carry is set, the temporary register is incremented; then the B accumulator is decremented. If the carry is clear, the B accumulator is immediately decremented. Since the B accumulator is not zero yet, the program loops back, and the process continues.

Notice that the A accumulator is rotated nine times. This is necessary since there are actually nine data bits including the carry bit, and the contents of this accumulator will be used by the main program. The carry bit will not affect the logic 1 bit count, since it was cleared by instruction  $7F_{16}$ .

After all of the bits in the A accumulator have been examined, and all 1's stored in the temporary register, the temporary register is rotated right. This places bit  $D_0$  in the carry bit position. the carry bit is then examined. If it was clear, the program counter returns to the main program; the ASCII code contains an even number of 1's, and does not require modification. If the carry bit was set, accumulator A is OR'ed with  $80_{16}$ , which places a 1 in the eighth (MSB) bit of the ASCII code. This makes the total number of 1's even in count. The program counter then returns to the main program.

To convert the parity bit routine to odd parity recognition, the code at address  $0071_{16}$  is changed to  $25_{16}$ . This causes a branch if the carry is set.

## Procedure (continued)

22. Pull the circuit timing wire from the 1 Hz socket and connect it to the LINE socket.
23. Press a number of Trainer keys and observe the serial transfer at data LED0.

## Discussion

When you press each Trainer key, you can see data LED0 flash on and off at a rapid rate. This is because the baud rate is now equal to the line frequency. Although the transfer rate appears to be quite fast, it is considered very slow for computer work. Typical speeds for a teletypewriter are almost twice as fast as the line frequency. Speeds as high as 96.2 kilobaud are quite common.

## Procedure (continued)

24. Do not disturb the circuit wired to your Trainer. It will be used in the next experiment. Proceed to Experiment 9.

## Experiment 9

### DIGITAL-TO-ANALOG AND ANALOG-TO-DIGITAL CONVERSION

#### OBJECTIVES:

*Show how to connect a digital-to-analog converter (DAC) to a microprocessor system.*

*Demonstrate how a DAC converts digital information into an analog equivalent.*

*Demonstrate a number of programs that will produce variable analog signal levels from a DAC.*

*Show how to convert a DAC into an analog-to-digital converter with a voltage comparator.*

*Demonstrate a program that implements the analog-to-digital conversion.*

#### Introduction

When analog input and output capabilities are added to a microprocessor, its power can be greatly expanded. Basically, the microprocessor is a digital device that is ideal for control of discrete input/output levels. However, many analog signals can also be processed with a minimum of additional hardware. With this addition, such devices as temperature sensors and photo cells can be monitored, and analog signals can be coupled to various peripherals such as oscilloscopes and audio amplifiers.

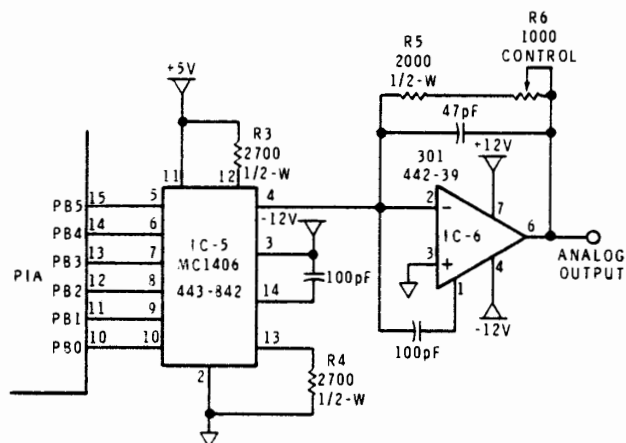
In this experiment, you will learn how to use the digital-to-analog converter (DAC) for outputting an analog signal. You will also learn how to translate an analog signal into its equivalent digital value using the same DAC.

## Material Required

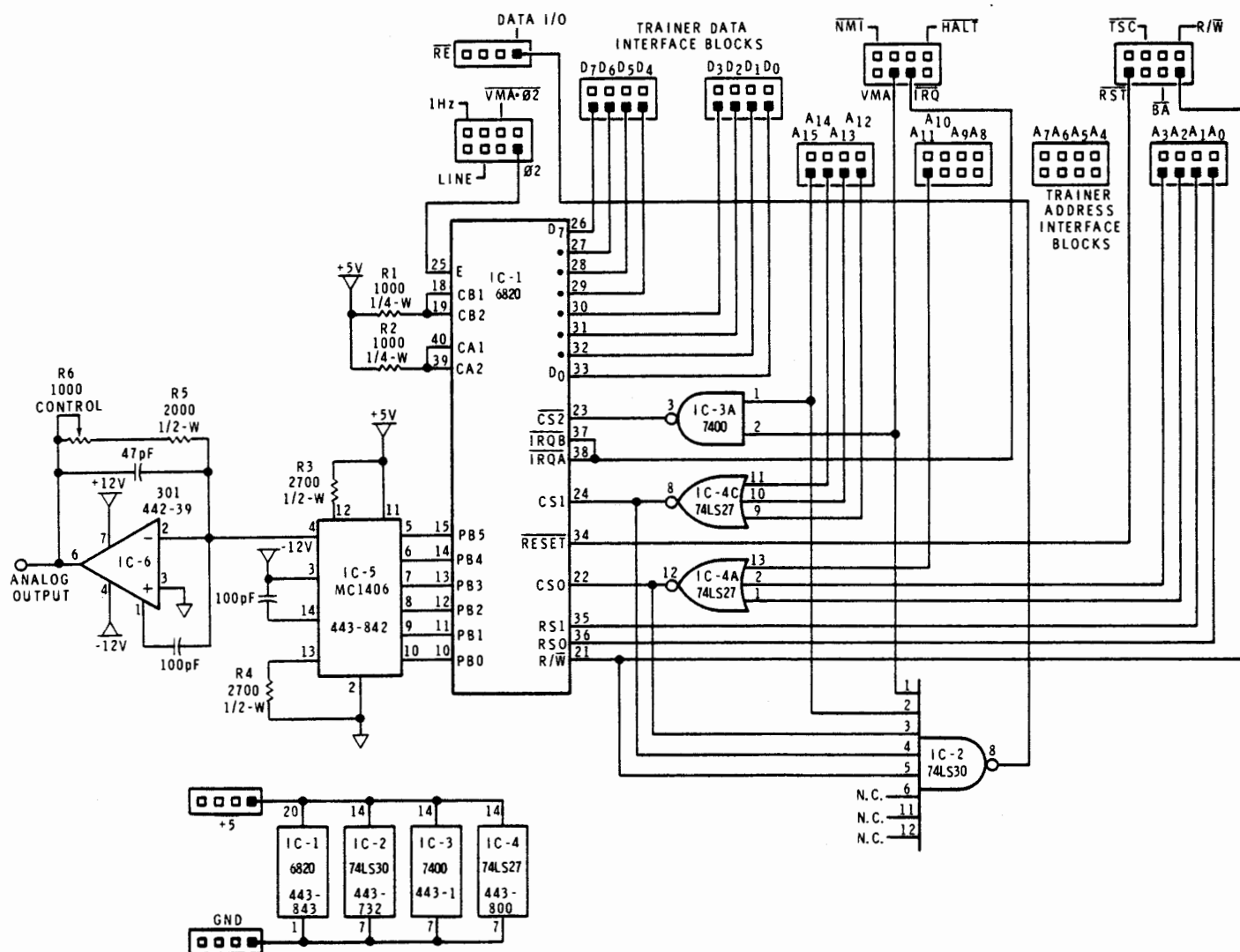
- 1 ET-3400 Microprocessor Trainer with the PIA circuit wired to its large connector block
- 3 1000 ohm, 1/4-watt, 10% resistors
- 1 2000 ohm, 1/2-watt, 5% resistor
- 2 2700 ohm, 1/2-watt, 5% resistors
- 2 10k ohm, 1/2-watt, 5% resistors
- 1 1M ohm, 1/2-watt, 5% resistor
- 2 1000 ohm controls
- 1 47 pF ceramic capacitor
- 3 100 pF ceramic capacitors
- 1 1N4149 diode (56-56)
- 1 5.6-volt zener diode (56-616)
- 1 741 op amp integrated circuit (442-22)
- 1 301 op amp integrated circuit (442-39)
- 1 MC1406 DAC integrated circuit (443-842)
- 1 VOM, VTVM, or DVM with input impedance greater than 100 k ohm (11 M ohm desirable)
- 1 Oscilloscope (optional)

## Procedure

1. Switch the Trainer power off. Then remove the wires interconnecting LINE and data LED7, data LED7 and PIA pin 2, and data LED0 and PIA pin 10. The remaining PIA circuitry will be used in this experiment.
2. Refer to Figure 10-77 and construct the circuit shown, on the large connector block affixed to the Trainer cabinet. Be careful when you insert 1/2-watt resistor leads. It is easy to push a connector strip out of the bottom of the block. The 1000 ohm control leads will fit into the connector block if you straighten them out and then insert the control at a slight angle. You can also solder a short wire to each control lead. Figure 10-78 shows the complete PIA circuit with the DAC circuit added.



**Figure 10-77**  
 DAC circuit added to the PIA interface  
 circuit.



**Figure 10-78**  
Circuit diagram for the digital-to-analog conversion circuit.

3. Switch the Trainer power on. Then enter the program listed in figure 10-79. Notice that the program begins at address 0100<sub>16</sub>.
4. Execute the program. Then, using the Trainer keyboard, press the 3 key and then the F key.
5. Set your voltmeter to measure 5 volts DC. Then connect the common lead to circuit ground, and the input lead to pin 6 of IC6 (analog output). You should measure approximately 0 volts DC.
6. Press 00 with the keyboard. Then adjust control R6 for a 5-volt output level. If you can not obtain a 5-volt level: switch the Trainer power off, change the value of resistor R5 from 2000 ohms to 1000 ohms, and then switch the Trainer power on again.
7. Press 20. What is the voltage level? \_\_\_\_\_
8. Press 30. What is the voltage level? \_\_\_\_\_
9. Press 10. What is the voltage level? \_\_\_\_\_

```

00001          NAM      LINOUT  REV. 0.1
00002          OPT      NOP
00003          FCBC     REDIS    EQU    $FCBC
00004          FE09     IHB      EQU    $FE09
00005 0100          ORG      $0100
00006          *CALIBRATION PROGRAM
00007 0100 CE FF04     LDX      $$FF04
00008 0103 FF 8002     STX      $8002      B SIDE IS OUT
00009 0106 BD FCBC NEWIN JSR      REDIS
00010 0109 BD FE09     JSR      IHB
00011 010C B7 8002     STA A    $8002
00012 010F 20 F5      BRA      NEWIN
00013          END

```

Figure 10-79

Program to convert digital values to  
their analog equivalent.

## Discussion

The DAC converts a binary word into a proportional output current. This particular DAC is a 6-bit device. That means it can accommodate a 6-bit binary word. Therefore, the two most significant data bits from the PIA are not used.

This DAC requires a complemented input, that is, maximum current out occurs when the PIA output is  $00_{16}$ . Minimum current occurs when all of the data lines are at a logic 1 level. This equals  $3F_{16}$  (data bits PB6, PB7 not used).

Op amp IC6 (301) functions as a current-to-voltage converter. Feedback through resistor R5 and control R6 determines the gain of the op amp.

Since the DAC accommodates a 6-bit binary word, it is possible to have a conversion resolution of  $2^6$  or 64 discrete current levels. With IC6 calibrated for a voltage swing between 0 and 5 volts, each binary unit equals approximately 0.08 volts DC. Therefore, the approximate voltage levels you should have observed in steps 7, 8, and 9 are:

$$20_{16} = 2.44 \text{ VDC}$$

$$30_{16} = 1.16 \text{ VDC}$$

$$10_{16} = 3.72 \text{ VDC}$$

Refer to the program shown in Figure 10-79. The first two instructions set up the PIA so the B side operates as an output. REDIS then points to display H for data display. A second monitor routine, IHB, couples key closure data to the display, and also loads the data into the A accumulator. Two key entries are required to complete the monitor routine. The data is then stored to the PIA. Since only the first six data bits are used by the DAC,  $00_{16}$ ,  $40_{16}$ , and  $C0_{16}$  will convert to the same analog levels.

## Procedure (continued)

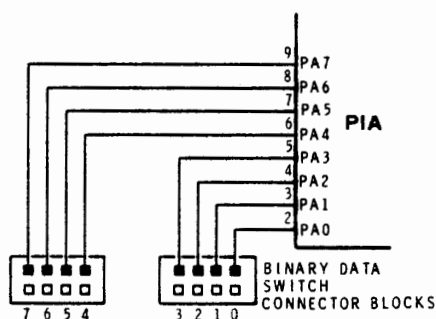


Figure 10-80

Peripheral data entry circuit for use with the digital-to-analog circuit.

10. Switch the Trainer power off. Then refer to Figure 10-80 and interconnect the PIA and the binary data switches.
11. Switch the Trainer power on. Then enter the program listed in Figure 10-81. Notice that the program begins at address 0120<sub>16</sub>.
12. Execute the program located at address 0120<sub>16</sub>. Then set the binary data switches for 1F<sub>16</sub>. The voltmeter should indicate a slowly decreasing voltage that cyclically ramps from 5 volts to 0 volts.
13. The rate of voltage change is determined by the binary data switches. If you have an oscilloscope, connect its input to pin 6 of IC6 and circuit ground. Now change the binary data switches to 00<sub>16</sub>. You should observe a descending voltage ramp.

## Discussion

The voltage ramp in this program is a composite of 64 discrete voltage steps. Each step represents a binary value. Therefore, an 8-bit DAC would have produced a ramp with 256 discrete steps, while a 4-bit DAC would contain only 16 steps.

00001		NAM	OUTRAMP1 REV. 0.1
00002		OPT	NOF
00003	0120	ORG	\$0120
00004	0120 CE 0004	LIX	##0004
00005	0123 FF 8000	STX	\$8000 A SIDE IN
00006	0126 CE FF04	LIX	##FF04
00007	0129 FF 8002	STX	\$8002 B SIDE OUT
00008	012C 4F	CLR A	
00009	012D B7 8002 NXTRMP	STA A	\$8002
00010	0130 FE 8000	LIX	\$8000 TIME TO WAIT
00011	0133 4C	INC A	INCR RAMP STEP
00012	0134 09	STHOLD DEX	
00013	0135 26 FD	BNE	STHOLD
00014	0137 20 F4	BRA	NXTRMP
00015		END	

Figure 10-81

Program to generate a voltage ramp.



## Procedure (continued)

14. Change the data at address  $0133_{16}$  to  $4A_{16}$ . Then execute the program. You should observe an ascending voltage ramp. If you only have a voltmeter, change the binary data switches to  $1F_{16}$ . This will slow the ramp rate so you can observe the voltage change.

## Discussion

The ramp program sets the A side of the PIA as an input, and the B side as an output. The A accumulator is cleared so the ramp will begin at +5 volts. Accumulator A is stored to the DAC through the PIA. Then the index register is loaded with the binary switch data. This will determine the waiting period between voltage ramp steps.

Remember that the index register always contains two bytes of data. The first byte (high byte) is obtained from the specified address, and the second byte (low byte) is obtained from the specified address plus one. Therefore, when you load the index register from the PIA, the high byte contains the binary switch data, and the low byte contains  $04_{16}$ .  $04_{16}$  is the data stored in the A side control register of the PIA during initialization.

Accumulator A is incremented for the next voltage level, and the index register is decremented until it reaches  $0000_{16}$ . Then the program branches back to the store the A accumulator, and the cycle continues. It is not necessary to clear accumulator A once the voltage ramp reaches zero, since the next increment sets the six least significant binary data bits to zero.

## Procedure (continued)

15. Enter the program listed in Figure 10-82. Notice that this program begins at address 0140<sub>16</sub>.
16. Execute the program beginning at address 0140<sub>16</sub>. Set the binary data switches to 1F<sub>16</sub>. The voltage should step from 0 to 5 volts and then back to 0 volts in a cyclic manner. You can observe this dual ramp (triangle) waveform on the oscilloscope, if you increase the ramp rate with the binary data switches.

## Discussion

This program uses the A side of the PIA to enter step delay data and the B side to output the discrete voltage level data. However, this time the A accumulator is loaded with 3F<sub>16</sub>, which is the binary equivalent of 0 volts.

00001		NAM	TRIANGLE REV. 0.1
00002		OPT	NOF
00003	0140	ORG	\$0140
00004	0140 CE 0004	LDX	#\$0004
00005	0143 FF 8000	STX	\$8000 A SIDE IN
00006	0146 CE FF04	LDX	#\$FF04
00007	0149 FF 8002	STX	\$8002 B SIDE OUT
00008	014C 86 3F	LDA A	#\$3F DAC OUT = 0
00009	014E FE 8000 UP	LDX	\$8000 DELAY TIME
00010	0151 B7 8002	STA A	\$8002 OUT TO DAC
00011	0154 27 10	BEQ	DOWN1
00012	0156 4A UP1	DEC A	
00013	0157 09 LOOP1	DEX	
00014	0158 26 FD	BNE	LOOP1
00015	015A 20 F2	BRA	UP
00016	015C FE 8000 DOWN	LDX	\$8000 DELAY TIME
00017	015F B7 8002	STA A	\$8002 OUT TO DAC
00018	0162 81 3F	CMP A	#\$3F
00019	0164 27 F0	BEQ	UP1
00020	0166 4C DOWN1	INC A	
00021	0167 09 LOOP2	DEX	
00022	0168 26 FD	BNE	LOOP2
00023	016A 20 F0	BRA	DOWN
00024		END	

Figure 10-82

Program to generate a triangle waveform.

Beginning at address  $014E_{16}$ , the index register is loaded with the level step delay time. Then the A accumulator is stored to the PIA. If accumulator A is zero, the program branches to address  $0166_{16}$ . Since it equals  $3F_{16}$ , the A accumulator is decremented. Then, the index register is decremented until it equals  $0000_{16}$ .

At the end of delay, the index register is again loaded, and the contents of the A accumulator are stored to the PIA. This cycle continues until the A accumulator equals  $00_{16}$  (5-volt level). Then the program branches to address  $0166_{16}$ .

Addresses  $015C$  through  $016B_{16}$  are similar to the first part of the program. They differ in that the A accumulator is incremented and compared to  $0F_{16}$ . Thus, the voltage level is cycled from 0 to 5 volts in the UP program section, and from 5 to 0 volts in the DOWN program section.

## Procedure (continued)

17. Enter the program listed in Figure 10-83. Notice that this program begins at address  $0000_{16}$ .
18. Execute the program. This program produces a sine waveform at a fixed frequency. If you only have a voltmeter, it should indicate approximately 1.7 volts AC. An oscilloscope will show a slightly distorted waveform. This is because of the resolution provided by the 6-bit DAC. An 8-bit DAC would produce a more symmetrical waveform.

## Discussion

The program uses a "look-up" table (addresses  $003B$  through  $004C_{16}$ ) of constant values to generate a sine wave signal. The table was produced by deriving the sine of the angles between  $0^\circ$  and  $90^\circ$  in  $5^\circ$  increments, and then multiplying each value by a constant.

Because the sine wave must reside within a 0 to 5-volt "window," each  $90^\circ$  segment of the waveform can only be generated by 32 of the possible data bits. The first  $90^\circ$  of the sine wave starts at approximately 2.5 volts and steps to 0 volts. The second  $90^\circ$  steps from 0 volts up to 2.5 volts. The third  $90^\circ$  steps from 2.5 volts up to 5 volts. Finally, the fourth  $90^\circ$  steps from 5 volts down to 2.5 volts.

```

00001      NAM      SINEWAVE REV.0.2
00002      OPT      NOP
00003 0000 CE FF04      LDX      #$FF04
00004 0003 FF 8002      STX      $8002
00005 0006 CE 003B      LDX      #BTABLE
00006 0009 C6 11      SIN1      LDA B      #$11
00007 000B A6 00      SIN1A     LDA A      X
00008 000D 01      NOP
00009 000E B7 8002      STA A      $8002
00010 0011 08      INX
00011 0012 5A      DEC B
00012 0013 26 F6      BNE      SIN1A
00013 0015 C6 11      SIN2      LDA B      #$11
00014 0017 A6 00      SIN2A     LDA A      X
00015 0019 01      NOP
00016 001A B7 8002      STA A      $8002
00017 001D 09      DEX
00018 001E 5A      DEC B
00019 001F 26 F6      BNE      SIN2A
00020 0021 C6 11      SIN3      LDA B      #$11
00021 0023 A6 00      SIN3A     LDA A      X
00022 0025 43      COM A
00023 0026 B7 8002      STA A      $8002
00024 0029 08      INX
00025 002A 5A      DEC B
00026 002B 26 F6      BNE      SIN3A
00027 002D C6 11      SIN4      LDA B      #$11
00028 002F A6 00      SIN4A     LDA A      X
00029 0031 43      COM A
00030 0032 B7 8002      STA A      $8002
00031 0035 09      DEX
00032 0036 5A      DEC B
00033 0037 26 F6      BNE      SIN4A
00034 0039 20 CE      BRA      SIN1
00035 003B 20      BTABLE FCB      $20,$22,$25,$27,$2A,$2D,$2F
      003C 22
      003D 25
      003E 27
      003F 2A
      0040 2D
      0041 2F
00036 0042 31      FCB      $31,$33,$35,$37,$39,$3A,$3B
      0043 33
      0044 35
      0045 37
      0046 39
      0047 3A
      0048 3B
00037 0049 3C      FCB      $3C,$3D,$3E,$3E
      004A 3D
      004B 3E
      004C 3E
00038      END

```

Figure 10-83  
Program to generate a sine waveform.

With the program using only 32 data levels for each waveform segment, the first five data bits in each byte determine the digital voltage level, while the sixth bit ( $D_5$ ) determines whether the lower two waveform segments or the upper two waveform segments are being generated. Refer to the program in Figure 10-83. Notice that the table of values begins at  $20_{16}$ , which approximately equals 2.5 volts. The values increase (voltage decrease) from there. When the upper half of the sine waveform is produced, the table values are complemented to generate the voltages between 2.5 and 5 volts.

In the program, the first two steps initialize the PIA. Then, the index register is loaded with the address of the first value in the look-up table. Accumulator B is loaded with  $11_{16}$ , which will be decremented to show when all of the table values have been used.

The next six instructions form a "fetch and display" loop that generates the first waveform segment. Accumulator A is loaded with the first value ( $20_{16}$ ). The NOP occupies time to make the loop time identical to the time used when the third and fourth segments are generated. Accumulator A is stored to the PIA. The index register is incremented, to point to the next value in the table, and the B accumulator is decremented. Since the register is not zero, the program branches back to address  $000B_{16}$ . This continues until the B accumulator equals  $00_{16}$ .

Then, the B accumulator is loaded with  $11_{16}$ . Remember that the index register contains the address of the last value in the table ( $004C_{16}$ ). The next six instructions form another fetch and display loop to generate the second waveform segment. The only instruction that differs from the first loop is "decrement the index register" rather than increment.

After the loop is completed, the B accumulator is loaded with  $11_{16}$ . The index register now contains the address ( $003B_{16}$ ) of the value at the top of the table. The next six instructions form the third fetch and display loop. This is identical to the first program loop; except, the NOP instruction is replaced with a "complement the A accumulator" instruction. Thus, the table values can now be used to generate the upper half of the sine waveform. Program loop four also operates in this manner.

## Procedure (continued)

19. Switch the Trainer power off. Then remove the eight wires interconnecting the binary data switches and the PIA.

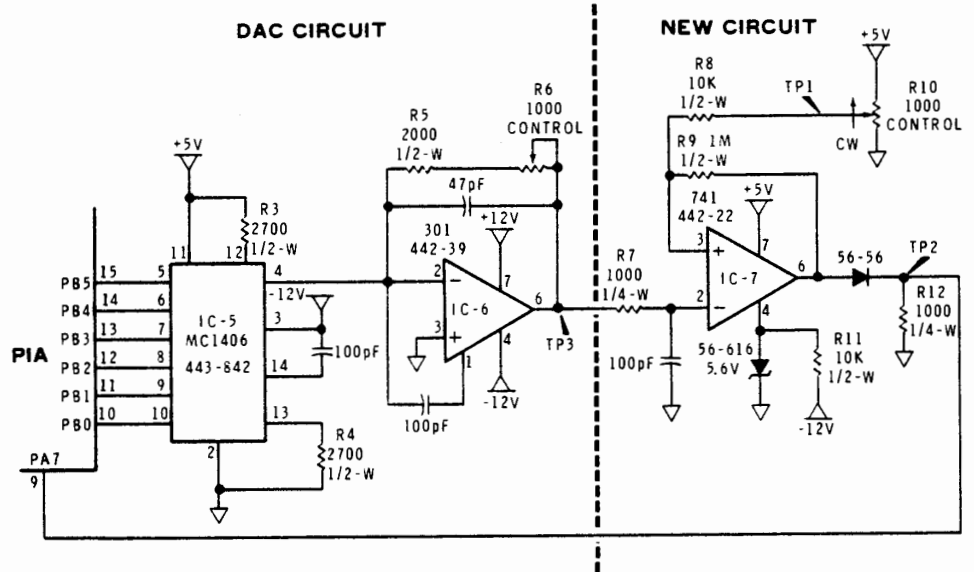


Figure 10-84

New circuit adds a voltage comparator to produce an analog-to-digital converter.

20. Refer to Figure 10-84 and add the new circuit shown to the DAC circuit. The output of the new circuit is connected to pin 9 (PA7) of the PIA. If you changed the value of R5 (in the DAC circuit) to 1000 ohms, in a previous section of this experiment, remove the 1000 ohm resistor and reinstall the 2000 ohm, 1/2-watt resistor in its place.
21. Switch the Trainer power on. Then, enter the program listed in Figure 10-85. Notice that the program begins at address 0170<sub>16</sub>.
22. Turn control R10 fully clockwise, then execute the program beginning at address 0170<sub>16</sub>.
23. Connect your voltmeter to TP1 (wiper of control R10) and circuit ground. Then, adjust control R6 for a Trainer display equal in value to the voltage at TP1. If you cannot adjust the display value down to the indicated voltage level, place a 1000 ohm, 1/4-watt resistor in series with resistor R5. Make sure you switch the Trainer power off before you add the resistor.
24. The circuit you have constructed acts like a digital voltmeter and will measure the voltage at the wiper of control R10 (TP1). Turn control R10 and compare the voltage indicated by your voltmeter with the voltage indicated by the Trainer display.

```
00001          NAM      A-TO-D      REV.0.1
00002          OPT      NOP
00003 0170          ORG      $0170
00004          FCBC      REDIS EQU    $FCBC
00005          FE20      OUTBYT EQU   $FE20
00006          *INITIALIZE PIA
00007 0170 CE 0004          LDX      #$0004
00008 0173 FF 8000          STX      $8000      A SIDE IN
00009 0176 CE FF04          LDX      #$FF04
00010 0179 FF 8002          STX      $8002      B SIDE OUT
00011          *FIND EQUAL POINT
00012 017C C6 FF      NEWIN LDA B      $FF      FF=LOW VOLTAGE
00013 017E 4F          CLR A
00014 017F F7 8002      NXTSTP STA B      $8002
00015 0182 CE 0055          LDX      #$0055      *
00016 0185 09          WAIT DEX          *HARDWARE SETTLE WAIT
00017 0186 26 FD          BNE      WAIT      *
00018 0188 7D 8000          TST      $8000      CHECK COMPARITOR
00019 018B 2A 06          BPL      FOUND
00020 018D 5A          DEC B
00021 018E 8B 01          ADD A      #01
00022 0190 19          DAA
00023 0191 20 EC          BRA      NXTSTP
00024          *OUTPUT RESULTS
00025 0193 BD FCBC      FOUND JSR      REDIS      SET DISPLAY LOCATION
00026 0196 BD FE20          JSR      OUTBYT      OUTPUT BYTE
00027 0199 86 01          LDA A      #01      * TURN ON
00028 019B B7 C16F          STA A      $C16F
00029 019E 20 DC          BRA      NEWIN      * DECIMAL POINT
00030          END
```

Figure 10-85

Program to convert an analog signal to  
a digital value.

## Discussion

In this analog-to-digital conversion circuit, the MPU supplies a known voltage to a voltage comparator, and looks for a match with the unknown voltage at the comparator. Because the program operates in digit increments, each voltage level step will equal 0.1 volts after the circuit has been calibrated. Thus, the unknown voltage can be resolved to one-tenth of a volt.

At the beginning of the program, the PIA is initialized (A side in, B side out); the B accumulator is loaded with FF (to start comparison at 0 volts); and the A accumulator is cleared (used to store the voltage count, for the display). The B accumulator is stored to the PIA for conversion to a

voltage level. The next three instructions use the index register to supply a short time delay. This is needed since the analog circuit cannot operate as quickly as the MPU.

After the short delay, the comparator output to the A side of the PIA is checked for a voltage match. If there is no match, the B accumulator is decremented, increasing the voltage level, and the A accumulator is incremented, by adding  $01_{16}$  to the contents, to indicate the voltage increase. Since this circuit is emulating a digital voltmeter, the displayed voltage must be in decimal (base 10). Therefore, the next instruction performs a decimal adjust on the A accumulator. This converts the binary addition to a BCD format.

The program then branches back to address  $017F_{16}$  and repeats until the comparator test indicates that the voltage generated by the MPU equals the unknown voltage. You can observe the conversion routine by connecting the Y1 input of a dual-trace oscilloscope to TP2 and circuit ground, and the Y2 input to TP3 and circuit ground. Figure 10-83 shows the location of the test points, while Figure 10-86 shows the display of the 0.5-volt conversion.

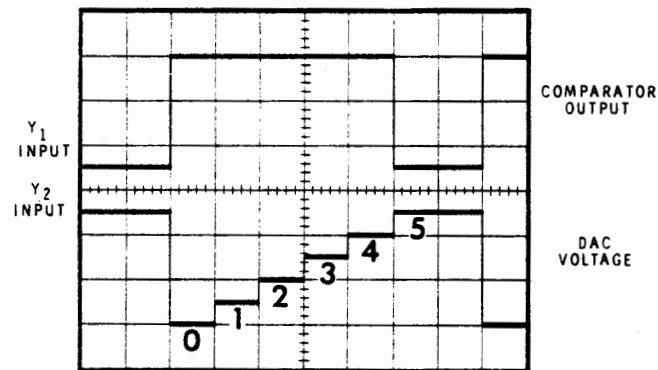


Figure 10-86  
Voltage comparator timing in relation  
to the DAC voltage signal.



As soon as the comparator senses that the known and unknown voltages are equal, it switches low to tell the MPU a match has occurred. This causes the program to branch to the display output routine. REDIS stores the location of display H. OUTBYT writes the contents of the A accumulator to displays H and I. Next, the contents of the A accumulator is changed to  $01_{16}$ . This is stored to address  $C16F_{16}$ , which lights the decimal point in display H.

## Procedure (continued)

25. Switch the Trainer power off. Then remove the wires and components from the two large connector blocks. Caution: The PIA is an MOS device. When you remove it from the Trainer, press it onto its conductive foam pad. This will reduce the possibility of damage from static electricity.
26. Return to the Unit Activity Guide of Unit 8.



# INDEX

## A

ADC (Add with Carry), 4-46 to 4-48, 9-81, 9-85 to 9-89  
 ADD (Add), 2-22 to 2-23, 2-36 to 2-39, 2-58 to 2-61, 4-29 to 4-32, 4-55, 9-20, 9-22 to 9-25, 9-31 to 9-33  
 ALU (Arithmetic Logic Unit), 2-16  
 AND 3-35 to 3-36, 9-36, 9-43 to 9-44  
 ANSCII (American National Standard Code for Information Interchange), 1-48  
 ASCII (American Standard Code for Information Interchange), 1-48 to 1-51  
 ASCII Code word format, 1-50, 10-91 to 10-93  
 ASLA (Arithmetic Shift Accumulator Left), 4-51 to 4-53, 9-90 to 9-94  
 Accumulator, 2-16 to 2-17, 2-64  
 Accumulator A, 5-8 to 5-9  
 Accumulator B, 5-8 to 5-9  
 Addend, 3-6  
 Address, 2-9, 2-24, 2-47, 7-33  
 Address bus, 2-18, 7-10, 10-14  
 Address clearing, 9-106  
 Address decoder, 2-18, 6-28, 7-7, 7-32 to 7-33, 8-10, 10-27 to 10-38  
 Address register, 2-17, 5-7  
 Addressable latch, 7-42 to 7-46  
 Addressing modes:  
     Combined, 2-66 to 2-67  
     Direct, 2-46 to 2-65  
     Extended, 5-9, 5-37 to 5-38, 9-102 to 9-103  
     Indexed, 5-39 to 5-45, 9-102, 9-104 to 9-108  
     Inherent (implied), 2-44, 2-46  
     Immediate, 2-45 to 2-48, 9-31  
     Relative, 4-7 to 4-8

Algorithms, 4-29 to 4-43

Alphanumeric codes, 1-48 to 1-53

Analog to digital conversions, 10-108 to 10-111

Arabic number system, See "Decimal Number System"

Arithmetic instructions, 5-16 to 5-18, "Appendix A"

Audio output, 10-71 to 10-82

Augend, 3-6

## B

BA (Bus Available), 7-12

BCC (Branch if Carry Clear), 4-26

BCD Codes (Binary Coded Decimal), 1-42 to 1-46, 4-37 to 4-43, 4-52 to 4-55, 9-92 to 9-99

BCS (Branch if Carry Set), 4-26, 4-43

BEQ (Branch if Equal Zero), 4-26, 9-55 to 9-58

BMI (Branch if Minus), 4-20 to 4-21, 4-26, 4-33, 4-35 to 4-36

BNE (Branch if Not Equal Zero), 4-26

BPL (Branch if Plus), 4-26

BRA (Unconditional Branch) (Branch Always), 4-20, 9-55, 9-57 to 9-60

BSR (Branch to Subroutine), 6-23

BVC (Branch if Overflow Clear), 4-26

BVS (Branch if Overflow Set), 4-26

Base (radix), 1-6

BAUDOT code, 1-52 to 1-53

Bi-directional bus, 7-6, 10-14

Binary arithmetic:

    Addition, 3-6 to 3-8, 3-26, 3-30 to 3-32, 5-42, 9-102 to 9-105 9-107 to 9-108

    Subtraction, 3-8 to 3-10, 9-114, 9-116 to 9-118

    Multiplication, 3-11 to 3-13

    Division, 3-14 to 3-15

Binary codes, 1-42 to 1-53  
Binary number system, 1-11 to 1-16, 1-42  
Binary point, 1-12 to 1-13  
Bit, 1-11, 2-10 to 2-11  
Boolean operations, 3-35 to 3-40  
Borrow, 3-9 to 3-10, 4-23  
Branch instruction execution, 4-8 to 4-13  
Branching, 4-6 to 4-17, 5-26 to 5-29, 9-45 to 9-79  
Branching backward, 4-16 to 4-17, 9-60 to 9-61  
Branching forward, 4-14 to 4-15, 9-59  
Buffer, 7-8 to 7-10  
Bus, 2-6, 7-6 to 7-8  
Byte, 2-10 to 2-11

## C

CE,  $\overline{CE}$  (Chip Enable), 7-7, 7-27 to 7-28, 7-30 to 7-33, 10-9  
CLI (Clear Interrupt Mask), 6-45  
CLRA (Clear Accumulator), 9-20, 9-26  
CMOS, 7-20  
CR (Control Register), 8-23  
CS (Chip Select Lines), 7-28, 8-28  
Carry, 3-6 to 3-8  
Carry flag (C), 4-23, 9-88 to 9-90  
Carry register, See "Carry Flag"  
Cascade stack, 6-6 to 6-9  
Chip, 2-3  
Clock signals ( $\phi 1$ ,  $\phi 2$ ), 7-11 to 7-14  
Condition codes, 4-22 to 4-26, 5-10, 5-30 to 5-31, 9-46 to 9-54  
Conditional branching, 4-20 to 4-26, 9-45, 9-67 to 9-69  
Conductors, 2-13  
Contact bounce, 8-7  
Contact bounce elimination, 8-16 to 8-17  
Contact closure detection, 8-6  
Controller sequencer, 2-18, 5-7

## Conversion:

BCD to binary, 4-37 to 4-40, 9-68 to 9-72  
Binary to BCD, 4-41 to 4-43  
Binary to decimal, 1-13, 9-6 to 9-8  
Binary to hexadecimal, 1-36 to 1-38  
Binary to octal, 1-24 to 1-26  
Decimal to binary, 1-14 to 1-16, 9-8 to 9-10  
Decimal to hexadecimal, 1-33 to 1-35, 9-12 to 9-16  
Decimal to octal, 1-21 to 1-23  
Hexadecimal to binary, 1-38 to 1-39  
Hexadecimal to decimal, 9-16 to 9-18  
Octal to binary, 1-26  
Octal to decimal, 1-20

## D

DAA (Decimal Adjust Accumulator), 4-53 to 4-55, 9-94 to 9-99  
DAC (Digital to Analog Conversion), 10-96 to 10-101  
DBE (Data Bus Enable), 7-10, 7-12  
DDR (Data Direction Register), 8-23  
DECA (Decrement Accumulator), 9-20, 9-26, 9-37 to 9-38  
DMA (Direct Memory Access), 7-10, 7-12  
Data bus, 2-19, 7-10  
Data handling instructions, 5-18 to 5-21, "Appendix A"  
Data register, 2-16, 2-17, 5-7  
Data storage, 9-29 to 9-30  
Data test instructions, 5-23 to 5-24, "Appendix A"  
Data transfer, 9-27 to 9-28  
Debounce, 10-60 to 10-64  
Decimal number system, 1-6 to 1-8  
Decoder-driver, 7-37 to 7-38

Decoders, See "Instruction Decoder"

Digital clock program, 9-138 to 9-146, 10-19 to 10-24

Displays, 7-36 to 7-41, 7-47 to 7-48, 8-33 to 8-36

Dividend, 3-14 to 3-15

Division, 4-33 to 4-36, 9-62 to 9-66

Divisor, 3-14 to 3-15

Do nothing instruction (NOP), see "NOP"

## E

ENCODE, 8-11 to 8-12, 8-16 to 8-17

8421 BCD code, See "BCD Codes"

exclusive OR, 3-38 to 3-39

Even parity, 1-50

Execute phase, 2-21, 2-34 to 2-36, 4-8 to 4-13, 7-14

## F

Fetch phase, 2-21, 2-29 to 2-33, 7-13 to 7-14

Flip-flops, 7-20, 7-22 to 7-23

Flow chart, 4-31, 4-34, 4-38, 4-42, 9-62, 9-90, 10-62

Fractional numbers, 1-7 to 1-8

## G

Gray code, 1-47

## H

HALT, 7-12

HLT (Halt), 2-22 to 2-23, 2-40

Half carry flag (H), 5-10

Hexadecimal number system, 1-29 to 1-39, 9-11 to 9-18

Higher order byte, 2-11

## I

INCA (Increment Accumulator), 9-20, 9-26

INCH (Input Character), 8-12, 8-16 to 8-17

IRQ (Interrupt Request), 6-44 to 6-45, 7-11 to 7-11, 8-22, 10-19 to 10-24

Improper timing, 10-10

Index register, 5-10, 5-24 to 5-25

Input, 2-7, 6-30 to 6-31, 7-22 to 7-23, 10-53 to 10-59

Input, 7-22 to 7-23

Input/output (I/O), 2-7, 6-27 to 6-34, 8-21 to 8-22, 10-67 to 10-70

Input/output device, 2-7, 6-28

Input/output interface, 6-28

Input/output port, 2-7

Input-output programming, 6-31 to 6-33

Input serial data, 10-88 to 10-89

Instruction, 2-8 "Appendix A"

Instruction decoder, 2-17, 5-7

Instruction set summary, 5-31, 5-46 to 5-48, "Appendix A"

Instruction timing, 7-13 to 7-14

Integer, 1-7

Interface lines, 7-10 to 7-12

Interfacing, 7-6 to 7-17

Interfacing requirements, 8-6 to 8-8

Interfacing with displays, 7-36 to 7-48, 10-40 to 10-42

Interfacing with switches, 8-6 to 8-17

Interfacing with RAM, 7-20 to 7-33

Interrupt, 6-34, 6-37 to 6-47, 10-59 to 10-64

Interrupt mask, 5-8, 5-10, 6-39, 6-45

Interrupt vector, 6-38

Invert, 3-40

## J

JMP (Jump), 4-6, 6-17 to 6-19

JSR (Jump to Subroutine), 6-20 to 6-21

## K

Key closure detection, 8-11 to 8-15

Key closure encoding, 8-11 to 8-15

Keyboard arrangement, 8-8 to 8-18

Keyboard circuit, 8-10 to 8-11

Keyboard decoding, 8-37 to 8-38

Keyboard switches, 8-6

**L**

LDA (Load Accumulator), 2-22 to 2-23, 9-20  
LED (Light Emitting Diodes), 6-29, 9-127 to 9-131  
    Common anode, 7-37  
    Common cathode, 7-37  
    Seven segment, 7-36 to 7-41, 10-43 to 10-52  
LSB (Least Significant Bit), 1-13 to 1-16, 1-24 to 1-26, 2-11  
LSD (Least Significant Digit), 1-8, 1-21 to 1-23, 1-33 to 1-35  
Location, 2-18  
Logic instructions, 5-22, "Appendix A"  
Looping, 4-6  
Lower order byte, 2-11

**M**

MOS, 7-20 to 7-21  
MOSFET, 7-21 to 7-22  
MPU (Microprocessor Unit), 2-6, 2-15 to 2-18, 7-30 to 7-31  
MPU cycle, 2-46, 7-13 to 7-17  
MSB (Most Significant Bit), 1-13 to 1-16, 1-24 to 1-26, 2-11  
MSD (Most Significant Digit), 1-8, 1-21 to 1-23, 1-33 to 1-35  
Memory, 2-18 to 2-20, 2-64, 10-8 to 10-18  
    (Memory) stack, 6-9 to 6-15  
Microcomputer, 2-6, 2-14  
Microprocessor, 2-3, 2-6, 2-14  
Microprocessor keyboard commands, 9-19  
Minuend, 3-8  
Mnemonic, 2-22  
Multiple-precision arithmetic, 4-47 to 4-50, 9-82 to 9-91, 9-94 to 9-97  
Multiplexing displays, 7-47 to 7-48, 8-33 to 8-36  
Multiplicand, 3-11 to 3-13  
Multiplication, 4-29 to 4-32, 4-51 to 4-53, 9-25 to 9-26, 9-32 to 9-33, 9-55 to 9-58, 9-92, 9-109 to 9-115  
Multiplier, 3-11 to 3-13

**N**

NDRO (Nondestructive Readout), 2-19  
NEGA (Complement 2's or Negate), 9-36, 9-39 to 9-40  
NMI (Nonmaskable Interrupt), 6-40 to 6-42, 7-10 to 7-11  
NOR (exclusive), 10-35 to 10-38  
NOP (Do Nothing Instruction), 9-46 to 9-47  
Negative Flag (N), 4-22  
Negative numbers, 3-16 to 3-21, 3-31 to 3-32, 9-37 to 9-40  
Negative powers of sixteen, B-53  
Negative powers of ten, 1-7  
Negative powers of two, 1-12, B-52  
Negative register, See "Negative Flag (N)"  
Nested subroutine, 6-22 to 6-23

**O**

OR, 3-36 to 3-38, 9-36, 9-43 to 9-44  
OR (Output Register), 8-23 to 8-27  
Octal number system, 1-20 to 1-26  
Odd parity, 1-50, 10-94  
Offset address, 5-40 to 5-41, 5-44 to 5-45, 9-107 to 9-108  
One's complement, 3-17 to 3-18  
Opcode, 2-22 to 2-24, 9-19, 9-33  
Operands, 2-16 to 2-17, 2-23 to 2-24, 2-46, 5-41  
Output, 2-7, 6-29 to 6-30, 10-40 to 10-52, 10-89 to 10-95  
Overflow, 1-15, 1-22, 1-33 to 1-34, 4-24 to 4-25  
Overflow flag (V), 4-24 to 4-25  
Overflow register, (See "Overflow Flag (V)")

**P**

Ø1 (Phi 1) (Clock Signal), 7-11, 7-13, 7-15 to 7-16  
Ø2 (Phi 2) (Clock Signal), 7-11 to 7-16, 7-32 to 7-33, 8-22  
PIA (Peripheral Interface Adapter), 8-20 to 8-39, 10-65 to 10-91  
PIA (continued)  
    Addressing, 8-28 to 8-30  
    Decoding keyboards, 8-37 to 8-38

- Decoding switch matrix, 8-38 to 8-39
- Driving seven segment displays, 8-33 to 8-36
- Initialization, 8-25 to 8-27
- Registers, 8-23 to 8-24
- Registers addressing, 8-24
- PULL, 6-8, 6-11 to 6-12, 9-122 to 9-123
- PUSH, 6-7, 6-10 to 6-11, 9-120 to 9-121
- Parity, 1-50, 10-93 to 10-95
- Pins, 7-10 to 7-12
- Positional notation:
  - Binary number system, 1-11 to 1-12
  - Decimal number system, 1-6 to 1-7
  - Hexadecimal number system, 1-32
  - Octal number system, 1-20
- Powers of eight, 1-20, B-53
- Powers of sixteen, 1-32, B-53
- Powers of ten, 1-6
- Powers of two, 1-11, B-51
- Program, 2-8, 10-16 to 10-17
- Program counter, 2-17, 5-9
- Program debugging, 9-73 to 9-79
- Program execution, 2-28 to 2-40, 2-52 to 2-65
- Program timing, 7-15 to 7-17
- Pure binary code, 1-42
- Q
- Quotient, 3-14 to 3-15
- R
- RAM (Random Access Memory), 2-20, 6-37, 7-6 to 7-7, 7-13, 7-20 to 7-33, 9-33 to 9-34
- RAM connection to MPU, 7-30 to 7-31
- RAM (Static), 7-20 to 7-23
- $\overline{RE}$  (Read Enable), 10-14 to 10-15
- READ, 2-19, 7-23, 7-27, 10-9
- RESET, 6-37 to 6-40, 7-10 to 7-11, 8-22, 8-25 to 8-26
- $\overline{RESET}$ , 6-38, 7-11
- ROM (Read Only Memory), 2-20, 6-37, 7-8, 7-13, 9-33 to 9-34
- RS (Register Select Line), 8-29 to 8-30
- RTI (Return From Interrupt), 6-42 to 6-43
- RTS (Return From Subroutine), 6-20 to 6-21
- $R/\overline{W}$  (Read/Write), 2-20, 7-10, 8-22, 10-9 to 10-10
- Radix, See "Base"
- Radix point, 1-8
- Read/write memory, 2-18, 2-20
- Reset interrupt sequence, 6-38
- S
- SBC (Subtract with Carry), 4-46, 4-49 to 4-50, 9-81, 9-89 to 9-90
- SEI (Set Interrupt Mask), 6-45
- STA (Store Accumulator), 2-62 to 2-65, 9-20, 9-23
- SUB (Subtract), 4-20 to 4-21, 4-23, 4-33 to 4-36, 9-36, 9-42
- SWI (Software Interrupt), 6-46 to 6-47
- Sign and magnitude, 3-16
- Signed number arithmetic, 3-30 to 3-31, 9-37 to 9-41
- Sine waveform, 10-105 to 10-107
- 6800 MPU:
  - Architecture, 5-6 to 5-12
  - Block diagram, 5-11 to 5-12
  - Data sheet, 7-17, "Appendix B"
  - Instruction set, 5-15 to 5-31, "Appendix A"
  - Interface lines, 7-10 to 7-12
  - Programming model, 5-8 to 5-10
- Software interrupt, 6-46 to 6-47
- Special binary codes, 1-47
- Square root program, 9-114 to 9-118
- Stack, 6-6, 6-13 to 6-15, 9-121 to 9-123
- Stack pointer, 5-10, 5-24 to 5-25, 6-9 to 6-10
- Standard noninverting buffer, 7-8
- Stored program concept, 2-8 to 2-9
- Straight line program, 4-6, 9-19 to 9-34
- Subroutine, 6-17 to 6-24, 9-127 to 9-146
- Subtrahend, 3-8 to 3-10
- Switch debouncing, 8-7, 10-60 to 10-64
- Switch decoding, 8-8
- Switch matrix decoding, 8-34 to 8-39, 10-84 to 10-87
- Switch selection, 8-6

**T**

TSC (Three state control line), 7-10 to 7-11  
Two's complement arithmetic, 3-27 to 3-29  
Three state logic, 7-8 to 7-10  
Three state noninverting buffer, 7-8 to 7-10, 8-6  
Timing, 10-10  
Triangle waveform, 10-104 to 10-105  
Two's complement, 3-18 to 3-21, 9-39  
Two's complement arithmetic, 3-26 to 3-27, 3-29  
to 3-30, 4-25

**U**

Unconditional branching, See "BRA"

**V**

VMA (Valid Memory Address), 7-12  
Voltage ramp, 10-102 to 10-103

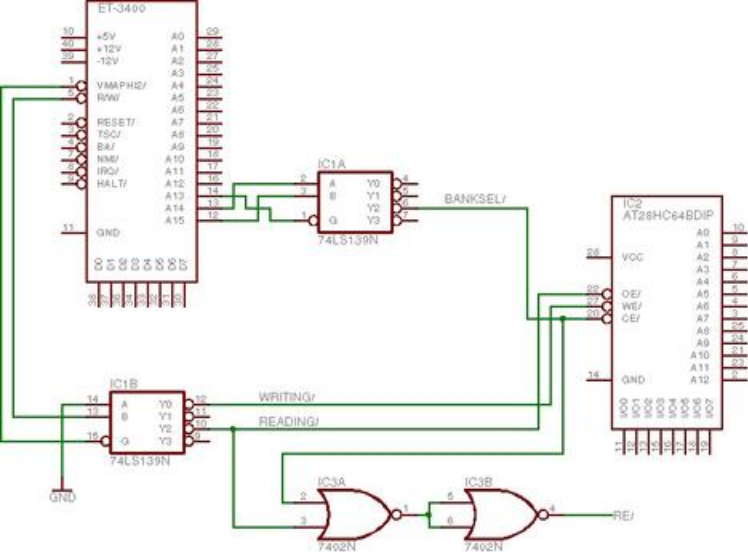
**W**

WAI (Wait for Interrupt), 6-47, 9-120 to 9-122  
WRITE, 2-20, 7-22 to 7-23, 7-27, 10-9  
Word, 2-9  
Word length, 2-9 to 2-11

**Z**

Zero flag (Z), 4-22  
Zero register, See "Zero Flag"

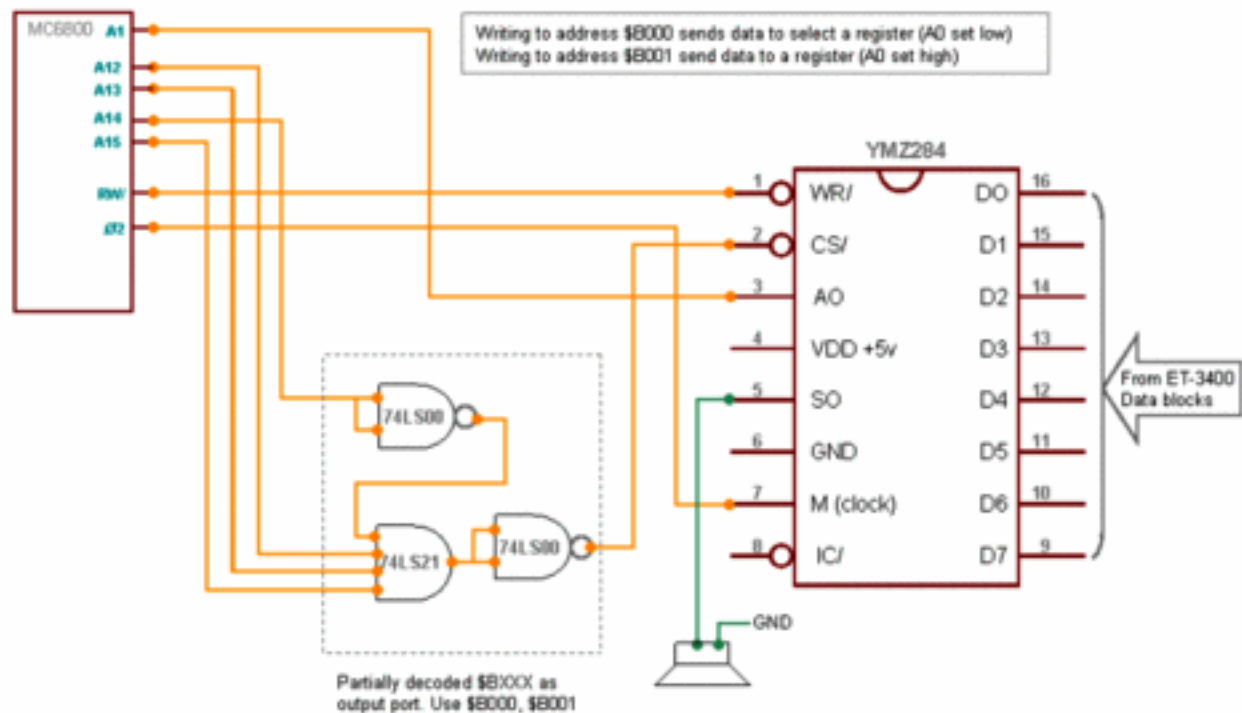












Heathkit ET-3400

View Images: normal | [\[full\]](#)

 [login](#)

Gallery: [6sys Gallery](#)  Album: [Electronics](#)  Album: [Heathkit ET-3400](#) 

1 of 3

[Next Photo](#)

[Last Photo](#)

Schematic for EEPROM board

Schematic for EEPROM board

1 of 3

[Next Photo](#)

[Last Photo](#)

Gallery: [6sys Gallery](#)  Album: [Electronics](#)  Album: [Heathkit ET-3400](#) 

Powered by [Gallery](#) v1





# Individual Learning Program

## MICROPROCESSORS

*Appendix A*

### DEFINITION OF THE EXECUTABLE INSTRUCTIONS

EE-3401

Courtesy of  
Motorola Semiconductor  
Products Inc.



# APPENDIX A

## Definition of the Executable Instructions

### A.1 Nomenclature

The following nomenclature is used in the subsequent definitions.

#### (a) Operators

( )	= contents of
←	= is transferred to
↑	= "is pulled from stack"
↓	= "is pushed into stack"
.	= Boolean AND
⊕	= Boolean (Inclusive) OR
⊕	= Exclusive OR
≈	= Boolean NOT

#### (b) Registers in the MPU

ACCA	= Accumulator A
ACCB	= Accumulator B
ACCX	= Accumulator ACCA or ACCB
CC	= Condition codes register
IX	= Index register, 16 bits
IXH	= Index register, higher order 8 bits
IXL	= Index register, lower order 8 bits
PC	= Program counter, 16 bits
PCH	= Program counter, higher order 8 bits
PCL	= Program counter, lower order 8 bits
SP	= Stack pointer
SPH	= Stack pointer high
SPL	= Stack pointer low

#### (c) Memory and Addressing

M	= A memory location (one byte)
M + 1	= The byte of memory at 0001 plus the address of the memory location indicated by "M."
Rel	= Relative address (i.e. the two's complement number stored in the second byte of machine code corresponding to a branch instruction).

#### (d) Bits 0 thru 5 of the Condition Codes Register

C	= Carry — borrow	bit — 0
V	= Two's complement overflow indicator	bit — 1
Z	= Zero indicator	bit — 2
N	= Negative indicator	bit — 3
I	= Interrupt mask	bit — 4
H	= Half carry	bit — 5

#### (e) Status of Individual Bits BEFORE Execution of an Instruction

An	= Bit n of ACCA (n=7,6,5,...,0)
Bn	= Bit n of ACCB (n=7,6,5,...,0)
IXHn	= Bit n of IXH (n=7,6,5,...,0)

IXLn = Bit n of IXL (n=7,6,5,...,0)

Mn = Bit n of M (n=7,6,5,...,0)

SPHn = Bit n of SPH (n=7,6,5,...,0)

SPLn = Bit n of SPL (n=7,6,5,...,0)

Xn = Bit n of ACCX (n=7,6,5,...,0)

(f) *Status of Individual Bits of the RESULT of Execution of an Instruction*

(i) For 8-bit Results

Rn = Bit n of the result (n =7,6,5,...,0)

This applies to instructions which provide a result contained in a single byte of memory or in an 8-bit register.

(ii) For 16-bit Results

RHn = Bit n of the more significant byte of the result  
(n =7,6,5,...,0)

RLn = Bit n of the less significant byte of the result  
(n =7,6,5,...,0)

This applies to instructions which provide a result contained in two consecutive bytes of memory or in a 16-bit register.

## **A.2 Executable Instructions (definition of)**

Detailed definitions of the 72 executable instructions of the source language are provided on the following pages.



**Add Accumulator B to Accumulator A**
**ABA**

 Operation:  $ACCA \leftarrow (ACCA) + (ACCB)$ 

Description: Adds the contents of ACCB to the contents of ACCA and places the result in ACCA.

 Condition Codes: H: Set if there was a carry from bit 3; cleared otherwise.  
 I: Not affected.  
 N: Set if most significant bit of the result is set; cleared otherwise.  
 Z: Set if all bits of the result are cleared; cleared otherwise.  
 V: Set if there was two's complement overflow as a result of the operation; cleared otherwise.  
 C: Set if there was a carry from the most significant bit of the result; cleared otherwise.

Boolean Formulae for Condition Codes:

$$H = A_3 \cdot B_3 + B_3 \cdot \bar{R}_3 + \bar{R}_3 \cdot A_3$$

$$N = R_7$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$V = A_7 \cdot B_7 \cdot \bar{R}_7 + \bar{A}_7 \cdot \bar{B}_7 \cdot R_7$$

$$C = A_7 \cdot B_7 + B_7 \cdot \bar{R}_7 + \bar{R}_7 \cdot A_7$$

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
Inherent	2	1	1B	033	027

## ADC

Add with Carry

Operation:  $ACCX \leftarrow (ACCX) + (M) + (C)$

Description: Adds the contents of the C bit to the sum of the contents of ACCX and M, and places the result in ACCX.

Condition Codes: H Set if there was a carry from bit 3; cleared otherwise.  
 I: Not affected.  
 N: Set if most significant bit of the result is set; cleared otherwise.  
 Z: Set if all bits of the result are cleared; cleared otherwise.  
 V: Set if there was two's complement overflow as a result of the operation; cleared otherwise.  
 C: Set if there was a carry from the most significant bit of the result; cleared otherwise.

Boolean Formulae for Condition Codes:

$$H = X_3 \cdot M_3 + M_3 \cdot \bar{R}_3 + \bar{R}_3 \cdot X_3$$

$$N = R_7$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$V = X_7 \cdot M_7 \cdot \bar{R}_7 + \bar{X}_7 \cdot \bar{M}_7 \cdot R_7$$

$$C = X_7 \cdot M_7 + M_7 \cdot \bar{R}_7 + \bar{R}_7 \cdot X_7$$

Addressing Formats:

See Table A-1

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

(DUAL OPERAND)

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A IMM	2	2	89	211	137
A DIR	3	2	99	231	153
A EXT	4	3	B9	271	185
A IND	5	2	A9	251	169
B IMM	2	2	C9	311	201
B DIR	3	2	D9	331	217
B EXT	4	3	F9	371	249
B IND	5	2	E9	351	233

## Add Without Carry

## ADD

Operation:  $ACCX \leftarrow (ACCX) + (M)$

Description: Adds the contents of ACCX and the contents of M and places the result in ACCX.

Condition Codes: H: Set if there was a carry from bit 3; cleared otherwise.  
 I: Not affected.  
 N: Set if most significant bit of the result is set; cleared otherwise.  
 Z: Set if all bits of the result are cleared; cleared otherwise.  
 V: Set if there was two's complement overflow as a result of the operation; cleared otherwise.  
 C: Set if there was a carry from the most significant bit of the result; cleared otherwise.

Boolean Formulae for Condition Codes:

$$H = X_3 \cdot M_3 + M_3 \cdot \bar{R}_3 + \bar{R}_3 \cdot X_3$$

$$N = R_7$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$V = X_7 \cdot M_7 \cdot \bar{R}_7 + \bar{X}_7 \cdot \bar{M}_7 \cdot R_7$$

$$C = X_7 \cdot M_7 + M_7 \cdot \bar{R}_7 + \bar{R}_7 \cdot X_7$$

Addressing Formats:

See Table A-1

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

(DUAL OPERAND)

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A IMM	2	2	8B	213	139
A DIR	3	2	9B	233	155
A EXT	4	3	BB	273	187
A IND	5	2	AB	253	171
B IMM	2	2	CB	313	203
B DIR	3	2	DB	333	219
B EXT	4	3	FB	373	251
B IND	5	2	EB	353	235

## AND

### Logical AND

Operation:  $ACCX \leftarrow (ACCX) \cdot (M)$

Description: Performs logical "AND" between the contents of ACCX and the contents of M and places the result in ACCX. (Each bit of ACCX after the operation will be the logical "AND" of the corresponding bits of M and of ACCX before the operation.)

Condition Codes: H: Not affected.  
 I: Not affected.  
 N: Set if most significant bit of the result is set; cleared otherwise.  
 Z: Set if all bits of the result are cleared; cleared otherwise.  
 V: Cleared.  
 C: Not affected.

Boolean Formulae for Condition Codes:

$$N = R_7$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$V = 0$$

Addressing Formats:

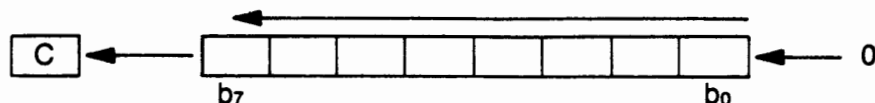
See Table A-1

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A IMM	2	2	84	204	132
A DIR	3	2	94	224	148
A EXT	4	3	B4	264	180
A IND	5	2	A4	244	164
B IMM	2	2	C4	304	196
B DIR	3	2	D4	324	212
B EXT	4	3	F4	364	244
B IND	5	2	E4	344	228

**Arithmetic Shift Left**
**ASL**

Operation:



**Description:** Shifts all bits of the ACCX or M one place to the left. Bit 0 is loaded with a zero. The C bit is loaded from the most significant bit of ACCX or M.

**Condition Codes:**

- H: Not affected.
- I: Not affected.
- N: Set if most significant bit of the result is set; cleared otherwise.
- Z: Set if all bits of the result are cleared; cleared otherwise.
- V: Set if, after the completion of the shift operation, EITHER (N is set and C is cleared) OR (N is cleared and C is set); cleared otherwise.
- C: Set if, before the operation, the most significant bit of the ACCX or M was set; cleared otherwise.

**Boolean Formulae for Condition Codes:**

$$N = R_7$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$V = N \oplus C = [N \cdot \bar{C}] \odot [\bar{N} \cdot C]$$

(the foregoing formula assumes values of N and C after the shift operation)

$$C = M_7$$

**Addressing Formats**

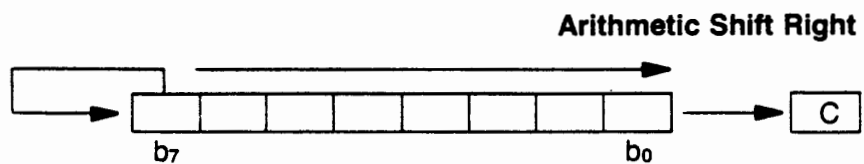
See Table A-3

**Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):**

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A	2	1	48	110	072
B	2	1	58	130	088
EXT	6	3	78	170	120
IND	7	2	68	150	104

## ASR

Operation:



Description: Shifts all bits of ACCX or M one place to the right. Bit 7 is held constant. Bit 0 is loaded into the C bit.

Condition Codes:

- H: Not affected.
- I: Not affected.
- N: Set if the most significant bit of the result is set; cleared otherwise.
- Z: Set if all bits of the result are cleared; cleared otherwise.
- V: Set if, after the completion of the shift operation, EITHER (N is set and C is cleared) OR (N is cleared and C is set); cleared otherwise.
- C: Set if, before the operation, the least significant bit of the ACCX or M was set; cleared otherwise.

Boolean Formulae for Condition Codes:

$$N = R_7$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$V = N \oplus C = [N \cdot \bar{C}] \odot [\bar{N} \cdot C]$$

(the foregoing formula assumes values of N and C after the shift operation)

$$C = M_0$$

Addressing Formats:

See Table A-3

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A	2	1	47	107	071
B	2	1	57	127	087
EXT	6	3	77	167	119
IND	7	2	67	147	103

**Branch if Carry Clear**
**BCC**

Operation:  $PC \leftarrow (PC) + 0002 + Rel$  if (C)=0

Description: Tests the state of the C bit and causes a branch if C is clear.

See BRA instruction for further details of the execution of the branch.

Condition Codes: Not affected.

Addressing Formats:

See Table A-8.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	24	044	036

## BCS

### Branch if Carry Set

Operation:  $PC \leftarrow (PC) + 0002 + Rel$  if (C)=1

Description: Tests the state of the C bit and causes a branch if C is set.

See BRA instruction for further details of the execution of the branch.

Condition Codes: Not affected.

Addressing Formats:

See Table A-8.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	25	045	037



**Branch if Equal**
**BEQ**

Operation:  $PC \leftarrow (PC) + 0002 + Rel$  if (Z)=1

Description: Tests the state of the Z bit and causes a branch if the Z bit is set.  
 See BRA instruction for further details of the execution of the branch.

Condition Codes: Not affected.

Addressing Formats:

See Table A-8.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	27	047	039

## BGE

**Branch if Greater than or Equal to Zero**

Operation:  $PC \leftarrow (PC) + 0002 + Rel$  if  $(N) \oplus (V) = 0$

i.e. if  $(ACCX) \geq (M)$

(Two's complement numbers)

Description: Causes a branch if (N is set and V is set) OR (N is clear and V is clear).

If the BGE instruction is executed immediately after execution of any of the instructions CBA, CMP, SBA, or SUB, the branch will occur if and only if the two's complement number represented by the minuend (i.e. ACCX) was greater than or equal to the two's complement number represented by the subtrahend (i.e. M).

See BRA instruction for details of the branch.

Condition Codes: Not affected.

Addressing Formats:

See Table A-8.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	2C	054	044

**Branch if Greater than Zero**
**BGT**

Operation:  $PC \leftarrow (PC) + 0002 + \text{Rel if } (Z) \odot [(N) \oplus (V)] = 0$

i.e. if  $(ACCX) > (M)$

(two's complement numbers)

Description: Causes a branch if [ Z is clear ] AND [(N is set and V is set) OR (N is clear and V is clear)].

If the BGT instruction is executed immediately after execution of any of the instructions CBA, CMP, SBA, or SUB, the branch will occur if and only if the two's complement number represented by the minuend (i.e. ACCX) was greater than the two's complement number represented by the subtrahend (i.e. M).

See BRA instruction for details of the branch.

Condition Codes: Not affected.

Addressing Formats:

See Table A-8.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	2E	056	046

## BHI

Branch if Higher

Operation:  $PC \leftarrow (PC) + 0002 + Rel$  if  $(C) \cdot (Z)=0$   
 i.e. if  $(ACCX) > (M)$   
 (unsigned binary numbers)

Description: Causes a branch if (C is clear) AND (Z is clear).

If the BHI instruction is executed immediately after execution of any of the instructions CBA, CMP, SBA, or SUB, the branch will occur if and only if the unsigned binary number represented by the minuend (i.e. ACCX) was greater than the unsigned binary number represented by the subtrahend (i.e. M).

See BRA instruction for details of the execution of the branch.

Condition Codes: Not affected.

Addressing Formats:

See Table A-8.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	22	042	034

**Bit Test**
**BIT**

Operation: (ACCX) · (M)

Description: Performs the logical "AND" comparison of the contents of ACCX and the contents of M and modifies condition codes accordingly. Neither the contents of ACCX or M operands are affected. (Each bit of the result of the "AND" would be the logical "AND" of the corresponding bits of M and ACCX.)

 Condition Codes: H: Not affected.  
 I: Not affected.  
 N: Set if the most significant bit of the result of the "AND" would be set; cleared otherwise.  
 Z: Set if all bits of the result of the "AND" would be cleared; cleared otherwise.  
 V: Cleared.  
 C: Not affected.

Boolean Formulae for Condition Codes:

$$N = R_7$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$V = 0$$

Addressing Formats:

See Table A-1.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A IMM	2	2	85	205	133
A DIR	3	2	95	225	149
A EXT	4	3	B5	265	181
A IND	5	2	A5	245	165
B IMM	2	2	C5	305	197
B DIR	3	2	D5	325	213
B EXT	4	3	F5	365	245
B IND	5	2	E5	345	229

## BLE

**Branch if Less than or Equal to Zero**

Operation:  $PC \leftarrow (PC) + 0002 + Rel$  if  $(Z) \odot [(N) \oplus (V)] = 1$

i.e. if  $(ACCX) \leq (M)$

(two's complement numbers)

Description: Causes a branch if [Z is set] OR [(N is set and V is clear) OR (N is clear and V is set)].

If the BLE instruction is executed immediately after execution of any of the instructions CBA, CMP, SBA, or SUB, the branch will occur if and only if the two's complement number represented by the minuend (i.e. ACCX) was less than or equal to the two's complement number represented by the subtrahend (i.e. M).

See BRA instruction for details of the branch.

Condition Codes: Not affected.

Addressing Formats:

See Table A-8.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	2F	057	047

### Branch if Lower or Same

# BLS

Operation:  $PC \leftarrow (PC) + 0002 + Rel$  if  $(C) \odot (Z) = 1$   
i.e. if  $(ACCX) \leq (M)$   
(unsigned binary numbers)

**Description:** Causes a branch if (C is set) OR (Z is set).

If the BLS instruction is executed immediately after execution of any of the instructions CBA, CMP, SBA, or SUB, the branch will occur if and only if the unsigned binary number represented by the minuend (i.e. ACCX) was less than or equal to the unsigned binary number represented by the subtrahend (i.e. M).

See BRA instruction for details of the execution of the branch.

**Condition Codes:** Not affected.

### Addressing Formats:

See Table A-8.

**Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):**

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	23	043	035

## BLT

Branch if Less than Zero

Operation:  $PC \leftarrow (PC) + 0002 + Rel$  if  $(N) \oplus (V) = 1$   
 i.e. if  $(ACCX) < (M)$   
 (two's complement numbers)

Description: Causes a branch if (N is set and V is clear) OR (N is clear and V is set).  
 If the BLT instruction is executed immediately after execution of any of the instructions CBA, CMP, SBA, or SUB, the branch will occur if and only if the two's complement number represented by the minuend (i.e. ACCX) was less than the two's complement number represented by the subtrahend (i.e. M).

See BRA instruction for details of the branch.

Condition Codes: Not affected.

Addressing Formats:

See Table A-8.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	2D	055	045



**Branch if Minus**
**BMI**

Operation:  $PC \leftarrow (PC) + 0002 + \text{Rel if } (N) = 1$

Description: Tests the state of the N bit and causes a branch if N is set.  
 See BRA instruction for details of the execution of the branch.

Condition Codes: Not affected.

Addressing Formats:

See Table A-8.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	2B	053	043

## BNE

Branch if Not Equal

Operation:  $PC \leftarrow (PC) + 0002 + Rel$  if  $(Z) = 0$

Description: Tests the state of the Z bit and causes a branch if the Z bit is clear.

See BRA instruction for details of the execution of the branch.

Condition Codes: Not affected.

Addressing Formats:

See Table A-8.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	26	046	038

# **BPL**

## **Branch if Plus**

Operation:  $PC \leftarrow (PC) + 0002 + Rel \text{ if } (N) = 0$

Description: Tests the state of the N bit and causes a branch if N is clear.  
 See BRA instruction for details of the execution of the branch.

Condition Codes: Not affected.

Addressing Formats:

See Table A-8.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	2A	052	042

## BRA

**Branch Always**

Operation:  $PC \leftarrow (PC) + 0002 + Rel$

Description: Unconditional branch to the address given by the foregoing formula, in which R is the relative address stored as a two's complement number in the second byte of machine code corresponding to the branch instruction.

Note: The source program specifies the destination of any branch instruction by its absolute address, either as a numerical value or as a symbol or expression which can be numerically evaluated by the assembler. The assembler obtains the relative address R from the absolute address and the current value of the program counter PC.

Condition Codes: Not affected.

Addressing Formats:

See Table A-8.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	20	040	032

**Branch to Subroutine**
**BSR**

Operation:  $PC \leftarrow (PC) + 0002$   
 $\downarrow (PCL)$   
 $SP \leftarrow (SP) - 0001$   
 $\downarrow (PCH)$   
 $SP \leftarrow (SP) - 0001$   
 $PC \leftarrow (PC) + Rel$

Description: The program counter is incremented by 2. The less significant byte of the contents of the program counter is pushed into the stack. The stack pointer is then decremented (by 1). The more significant byte of the contents of the program counter is then pushed into the stack. The stack pointer is again decremented (by 1). A branch then occurs to the location specified by the program.

See BRA instruction for details of the execution of the branch.

Condition Codes: Not affected.

Addressing Formats:

See Table A-8.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	8	2	8D	215	141

**BRANCH TO SUBROUTINE EXAMPLE**

		Memory Location	Machine Code (Hex)	Label	Assembler Language Operator	Operand
A. Before	PC	$\leftarrow$ \$1000	8D		BSR	CHARLI
			50			
	SP	$\leftarrow$ \$EFFF				
B. After	PC	$\leftarrow$ \$1052	**	CHARLI	***	*****
	SP	$\leftarrow$ \$EFFD				
		\$EFFE	10			
		\$EFFF	02			

## BVC

**Branch if Overflow Clear**

Operation:  $PC \leftarrow (PC) + 0002 + Rel$  if  $(V) = 0$

Description: Tests the state of the V bit and causes a branch if the V bit is clear.

See BRA instruction for details of the execution of the branch.

Condition Codes: Not affected.

Addressing Formats:

See Table A-8.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	28	050	040

**Branch if Overflow Set**
**BVS**

Operation:  $PC \leftarrow (PC) + 0002 + Rel$  if  $(V) = 1$

Description: Tests the state of the V bit and causes a branch if the V bit is set.

See BRA instruction for details of the execution of the branch.

Condition Codes: Not affected.

Addressing Formats:

See Table A-8.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
REL	4	2	29	051	041

## CBA

### Compare Accumulators

Operation: (ACCA) – (ACCB)

Description: Compares the contents of ACCA and the contents of ACCB and sets the condition codes, which may be used for arithmetic and logical conditional branches. Both operands are unaffected.

Condition Codes:

- H: Not affected.
- I: Not affected.
- N: Set if the most significant bit of the result of the subtraction would be set; cleared otherwise.
- Z: Set if all bits of the result of the subtraction would be cleared; cleared otherwise.
- V: Set if the subtraction would cause two's complement overflow; cleared otherwise.
- C: Set if the subtraction would require a borrow into the most significant bit of the result; clear otherwise.

Boolean Formulae for Condition Codes:

$$N = R_7$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$V = A_7 \cdot \bar{B}_7 \cdot \bar{R}_7 + \bar{A}_7 \cdot B_7 \cdot R_7$$

$$C = \bar{A}_7 \cdot B_7 + B_7 \cdot R_7 + R_7 \cdot \bar{A}_7$$

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	2	1	11	021	017



# CLC

## Clear Carry

Operation: C bit  $\leftarrow$  0

Description: Clears the carry bit in the processor condition codes register.

Condition Codes: H: Not affected.  
 I: Not affected.  
 N: Not affected.  
 Z: Not affected.  
 V: Not affected.  
 C: Cleared

Boolean Formulae for Condition Codes:

$$C = 0$$

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	2	1	0C	014	012

## CLI

### Clear Interrupt Mask

Operation: I bit  $\leftarrow$  0

Description: Clears the interrupt mask bit in the processor condition codes register. This enables the microprocessor to service an interrupt from a peripheral device if signalled by a high state of the "Interrupt Request" control input.

Condition Codes: H: Not affected.  
 I: Cleared.  
 N: Not affected.  
 Z: Not affected.  
 V: Not affected.  
 C: Not affected.

Boolean Formulae for Condition Codes:

$$I = 0$$

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	2	1	0E	016	014

**Clear**
**CLR**

 Operation:  $ACCX \leftarrow 00$ 

 or:  $M \leftarrow 00$ 

Description: The contents of ACCX or M are replaced with zeros.

Condition Codes: H: Not affected.

I: Not affected.

N: Cleared

Z: Set

V: Cleared

C: Cleared

Boolean Formulae for Condition Codes:

 $N = 0$ 
 $Z = 1$ 
 $V = 0$ 
 $C = 0$ 

Addressing Formats:

See Table A-3.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A	2	1	4F	117	079
B	2	1	5F	137	095
EXT	6	3	7F	177	127
IND	7	2	6F	157	111

## CLV

### Clear Two's Complement Overflow Bit

Operation:  $V \text{ bit} \leftarrow 0$

Description: Clears the two's complement overflow bit in the processor condition codes register.

Condition Codes: H: Not affected.  
 I: Not affected.  
 N: Not affected.  
 Z: Not affected.  
 V: Cleared.  
 C: Not affected.

Boolean Formulae for Condition Codes:  
 $V = 0$

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	2	1	0A	012	010

**Compare**
**CMP**

Operation: (ACCX) – (M)

Description: Compares the contents of ACCX and the contents of M and determines the condition codes, which may be used subsequently for controlling conditional branching. Both operands are unaffected.

 Condition Codes: H: Not affected.  
 I: Not affected.  
 N: Set if the most significant bit of the result of the subtraction would be set; cleared otherwise.  
 Z: Set if all bits of the result of the subtraction would be cleared; cleared otherwise.  
 V: Set if the subtraction would cause two's complement overflow; cleared otherwise.  
 C: Carry is set if the absolute value of the contents of memory is larger than the absolute value of the accumulator; reset otherwise.

Boolean Formulae for Condition Codes:

$$N = R_7$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$V = X_7 \cdot \bar{M}_7 \cdot \bar{R}_7 + \bar{X}_7 \cdot M_7 \cdot R_7$$

$$C = \bar{X}_7 \cdot M_7 + M_7 \cdot R_7 + R_7 \cdot \bar{X}_7$$

Addressing Formats:

See Table A-1.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

(DUAL OPERAND)

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A IMM	2	2	81	201	129
A DIR	3	2	91	221	145
A EXT	4	3	B1	261	177
A IND	5	2	A1	241	161
B IMM	2	2	C1	301	193
B DIR	3	2	D1	321	209
B EXT	4	3	F1	361	241
B IND	5	2	E1	341	225

# COM

**Complement**

 Operation:  $ACCX \leftarrow \approx (ACCX) = FF - (ACCX)$ 

 or:  $M \leftarrow \approx (M) = FF - (M)$ 

Description: Replaces the contents of ACCX or M with its one's complement. (Each bit of the contents of ACCX or M is replaced with the complement of that bit.)

 Condition Codes: H: Not affected.  
 I: Not affected.  
 N: Set if most significant bit of the result is set; cleared otherwise.  
 Z: Set if all bits of the result are cleared; cleared otherwise.  
 V: Cleared.  
 C: Set.

Boolean Formulae for Condition Codes:

$$N = R_7$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$V = 0$$

$$C = 1$$

Addressing Formats:

See Table A-3.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A	2	1	43	103	067
B	2	1	53	123	083
EXT	6	3	73	163	115
IND	7	2	63	143	099

## Compare Index Register

## CPX

Operation: (IXL) – (M+1)  
 (IXH) – (M)

Description: The more significant byte of the contents of the index register is compared with the contents of the byte of memory at the address specified by the program. The less significant byte of the contents of the index register is compared with the contents of the next byte of memory, at one plus the address specified by the program. The Z bit is set or reset according to the results of these comparisons, and may be used subsequently for conditional branching.

The N and V bits, though determined by this operation, are not intended for conditional branching.

The C bit is not affected by this operation.

Condition Codes: H: Not affected.  
 I: Not affected.  
 N: Set if the most significant bit of the result of the subtraction from the more significant byte of the index register would be set; cleared otherwise.  
 Z: Set if all bits of the results of both subtractions would be cleared; cleared otherwise.  
 V: Set if the subtraction from the more significant byte of the index register would cause two's complement overflow; cleared otherwise.  
 C: Not affected.

Boolean Formulae for Condition Codes:

$$N = RH_7$$

$$Z = (\overline{RH_7} \cdot \overline{RH_6} \cdot \overline{RH_5} \cdot \overline{RH_4} \cdot \overline{RH_3} \cdot \overline{RH_2} \cdot \overline{RH_1} \cdot \overline{RH_0}) \cdot (\overline{RL_7} \cdot \overline{RL_6} \cdot \overline{RL_5} \cdot \overline{RL_4} \cdot \overline{RL_3} \cdot \overline{RL_2} \cdot \overline{RL_1} \cdot \overline{RL_0})$$

$$V = IXH_7 \cdot \overline{M_7} \cdot \overline{RH_7} + \overline{IXH_7} \cdot M_7 \cdot RH_7$$

Addressing Formats:

See Table A-5.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
IMM	3	3	8C	214	140
DIR	4	2	9C	234	156
EXT	5	3	BC	274	188
IND	6	2	AC	254	172

## DAA

### Decimal Adjust ACCA

**Operation:** Adds hexadecimal numbers 00, 06, 60, or 66 to ACCA, and may also set the carry bit, as indicated in the following table:

State of C-bit before DAA (Col. 1)	Upper Half-byte (bits 4-7) (Col. 2)	Initial Half-carry H-bit (Col.3)	Lower to ACCA (bits 0-3) (Col. 4)	Number Added after by DAA (Col. 5)	State of C-bit DAA (Col. 6)
0	0-9	0	0-9	00	0
0	0-8	0	A-F	06	0
0	0-9	1	0-3	06	0
0	A-F	0	0-9	60	1
0	9-F	0	A-F	66	1
0	A-F	1	0-3	66	1
1	0-2	0	0-9	60	1
1	0-2	0	A-F	66	1
1	0-3	1	0-3	66	1

**Note:** Columns (1) through (4) of the above table represent all possible cases which can result from any of the operations ABA, ADD, or ADC, with initial carry either set or clear, applied to two binary-coded-decimal operands. The table shows hexadecimal values.

**Description:** If the contents of ACCA and the state of the carry-borrow bit C and the half-carry bit H are all the result of applying any of the operations ABA, ADD, or ADC to binary-coded-decimal operands, with or without an initial carry, the DAA operation will function as follows.

Subject to the above condition, the DAA operation will adjust the contents of ACCA and the C bit to represent the correct binary-coded-decimal sum and the correct state of the carry.

**Condition Codes:**

- H: Not affected.
- I: Not affected.
- N: Set if most significant bit of the result is set; cleared otherwise.
- Z: Set if all bits of the result are cleared; cleared otherwise.
- V: Not defined.
- C: Set or reset according to the same rule as if the DAA and an immediately preceding ABA, ADD, or ADC were replaced by a hypothetical binary-coded-decimal addition.

**Boolean Formulae for Condition Codes:**

$$N = R_7$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

C = See table above.



Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	2	1	19	031	025

# DEC

**Decrement**

 Operation:  $ACCX \leftarrow (ACCX) - 01$ 

 or:  $M \leftarrow (M) - 01$ 

Description: Subtract one from the contents of ACCX or M.

The N, Z, and V condition codes are set or reset according to the results of this operation.

The C bit is not affected by the operation.

Condition Codes: H: Not affected.

I: Not affected.

N: Set if most significant bit of the result is set; cleared otherwise.

Z: Set if all bits of the result are cleared; cleared otherwise.

V: Set if there was two's complement overflow as a result of the operation; cleared otherwise. Two's complement overflow occurs if and only if (ACCX) or (M) was 80 before the operation.

C: Not affected.

Boolean Formulae for Condition Codes:

$$N = R_7$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$V = X_7 \cdot \bar{X}_6 \cdot \bar{X}_5 \cdot \bar{X}_4 \cdot \bar{X}_3 \cdot \bar{X}_2 \cdot \bar{X}_0 = \bar{R}_7 \cdot R_6 \cdot R_5 \cdot R_4 \cdot R_3 \cdot R_2 \cdot R_1 \cdot R_0$$

Addressing Formats:

See Table A-3.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A	2	1	4A	112	074
B	2	1	5A	132	090
EXT	6	3	7A	172	122
IND	7	2	6A	152	106

**Decrement Stack Pointer**
**DES**

 Operation:  $SP \leftarrow (SP) - 0001$ 

Description: Subtract one from the stack pointer.

Condition Codes: Not affected.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	4	1	34	064	052

## DEX

### Decrement Index Register

Operation:  $IX \leftarrow (IX) - 0001$

Description: Subtract one from the index register.

Only the Z bit is set or reset according to the result of this operation.

Condition Codes: H: Not affected.  
 I: Not affected.  
 N: Not affected.  
 Z: Set if all bits of the result are cleared; cleared otherwise.  
 V: Not affected.  
 C: Not affected.

Boolean Formulae for Condition Codes:

$$Z = (\overline{RH_7} \cdot \overline{RH_6} \cdot \overline{RH_5} \cdot \overline{RH_4} \cdot \overline{RH_3} \cdot \overline{RH_2} \cdot \overline{RH_1} \cdot \overline{RH_0}) \cdot (\overline{RL_7} \cdot \overline{RL_6} \cdot \overline{RL_5} \cdot \overline{RL_4} \cdot \overline{RL_3} \cdot \overline{RL_2} \cdot \overline{RL_1} \cdot \overline{RL_0})$$

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	4	1	09	011	009

**Exclusive OR**
**EOR**

Operation:  $ACCX \leftarrow (ACCX) \oplus (M)$

Description: Perform logical "EXCLUSIVE OR" between the contents of ACCX and the contents of M, and place the result in ACCX. (Each bit of ACCX after the operation will be the logical "EXCLUSIVE OR" of the corresponding bit of M and ACCX before the operation.)

Condition Codes: H: Not affected.  
 I: Not affected.  
 N: Set if most significant bit of the result is set; cleared otherwise.  
 Z: Set if all bits of the result are cleared; cleared otherwise.  
 V: Cleared  
 C: Not affected.

Boolean Formulae for Condition Codes:

$$N = R_7$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$V = 0$$

Addressing Formats:

See Table A-1.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A IMM	2	2	88	210	136
A DIR	3	2	98	230	152
A EXT	4	3	B8	270	184
A IND	5	2	A8	250	168
B IMM	2	2	C8	310	200
B DIR	3	2	D8	330	216
B EXT	4	3	F8	370	248
B IND	5	2	E8	350	232

# INC

Increment

 Operation:  $ACCX \leftarrow (ACCX) + 01$ 

 or:  $M \leftarrow (M) + 01$ 

Description: Add one to the contents of ACCX or M.

The N, Z, and V condition codes are set or reset according to the results of this operation.

The C bit is not affected by the operation.

Condition Codes: H: Not affected.

I: Not affected.

N: Set if most significant bit of the result is set; cleared otherwise.

Z: Set if all bits of the result are cleared; cleared otherwise.

V: Set if there was two's complement overflow as a result of the operation; cleared otherwise. Two's complement overflow will occur if and only if (ACCX) or (M) was 7F before the operation.

C: Not affected.

Boolean Formulae for Condition Codes:

$$N = R_7$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$V = \bar{X}_7 \cdot X_6 \cdot X_5 \cdot X_4 \cdot X_3 \cdot X_2 \cdot X_1 \cdot X_0$$

$$C = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

Addressing Formats:

See Table A-3.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A	2	1	4C	114	076
B	2	1	5C	134	092
EXT	6	3	7C	174	124
IND	7	2	6C	154	108

# **Increment Stack Pointer**

**INS**

Operation:  $SP \leftarrow (SP) + 0001$

Description: Add one to the stack pointer.

Condition Codes: Not affected.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	4	1	31	061	049

# INX

## Increment Index Register

Operation:  $IX \leftarrow (IX) + 0001$

Description: Add one to the index register.

Only the Z bit is set or reset according to the result of this operation.

Condition Codes: H: Not affected.  
 I: Not affected.  
 N: Not affected.  
 Z: Set if all 16 bits of the result are cleared; cleared otherwise.  
 V: Not affected.  
 C: Not affected.

Boolean Formulae for Condition Codes:

$$Z = (\overline{RH_7} \cdot \overline{RH_6} \cdot \overline{RH_5} \cdot \overline{RH_4} \cdot \overline{RH_3} \cdot \overline{RH_2} \cdot \overline{RH_1} \cdot \overline{RH_0}) \cdot (\overline{RL_7} \cdot \overline{RL_6} \cdot \overline{RL_5} \cdot \overline{RL_4} \cdot \overline{RL_3} \cdot \overline{RL_2} \cdot \overline{RL_1} \cdot \overline{RL_0})$$

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	4	1	08	010	008



**Jump**
**JMP**

Operation: PC ← numerical address

Description: A jump occurs to the instruction stored at the numerical address. The numerical address is obtained according to the rules for EXTended or INDexed addressing.

Condition Codes: Not affected.

Addressing Formats:

See Table A-7.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
EXT	3	3	7E	176	126
IND	4	2	6E	156	110

# JSR

## Jump to Subroutine

Operation:

Either:  $PC \leftarrow (PC) + 0003$  (for EXTended addressing)

or:  $PC \leftarrow (PC) + 0002$  (for INDexed addressing)

Then:  $\downarrow$  (PCL)

$SP \leftarrow (SP) - 0001$

$\downarrow$  (PCH)

$SP \leftarrow (SP) - 0001$

$PC \leftarrow$  numerical address

**Description:** The program counter is incremented by 3 or by 2, depending on the addressing mode, and is then pushed onto the stack, eight bits at a time. The stack pointer points to the next empty location in the stack. A jump occurs to the instruction stored at the numerical address. The numerical address is obtained according to the rules for EXTended or INDexed addressing.

**Condition Codes:** Not affected.

**Addressing Formats:**

See Table A-7.

**Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):**

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
EXT	9	3	BD	275	189
IND	8	2	AD	255	173

### JUMP TO SUBROUTINE EXAMPLE (extended mode)

		Memory Location	Machine Code (Hex)	Label	Assembler Language Operator	Operand
A. Before:	PC	→	\$0FFF			
			\$1000			
			\$1001			
	SP	←	\$EFFF			
B. After:	PC	→	\$2077	CHARLI	***	*****
	SP	→	\$EFFD			
			\$EFFE			
			\$EFFF			

**Load Accumulator**
**LDA**

Operation:  $ACCX \leftarrow (M)$

Description: Loads the contents of memory into the accumulator. The condition codes are set according to the data.

Condition Codes: H: Not affected.  
 I: Not affected.  
 N: Set if most significant bit of the result is set; cleared otherwise.  
 Z: Set if all bits of the result are cleared; cleared otherwise.  
 V: Cleared.  
 C: Not affected.

Boolean Formulae for Condition Codes:

$$N = R_7$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$V = 0$$

Addressing Formats:

See Table A-1.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

(DUAL OPERAND)

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A IMM	2	2	86	206	134
A DIR	3	2	96	226	150
A EXT	4	3	B6	266	182
A IND	5	2	A6	246	166
B IMM	2	2	C6	306	198
B DIR	3	2	D6	326	214
B EXT	4	3	F6	366	246
B IND	5	2	E6	346	230

## LDS

### Load Stack Pointer

Operation:  $SPH \leftarrow (M)$

$SPL \leftarrow (M+1)$

Description: Loads the more significant byte of the stack pointer from the byte of memory at the address specified by the program, and loads the less significant byte of the stack pointer from the next byte of memory, at one plus the address specified by the program.

Condition Codes: H: Not affected.

I: Not affected.

N: Set if the most significant bit of the stack pointer is set by the operation; cleared otherwise.

Z: Set if all bits of the stack pointer are cleared by the operation; cleared otherwise.

V: Cleared.

C: Not affected.

Boolean Formulae for Condition Codes:

$N = RH_7$

$Z = (\overline{RH_7} \cdot \overline{RH_6} \cdot \overline{RH_5} \cdot \overline{RH_4} \cdot \overline{RH_3} \cdot \overline{RH_2} \cdot \overline{RH_1} \cdot \overline{RH_0}) \cdot$   
 $(\overline{RL_7} \cdot \overline{RL_6} \cdot \overline{RL_5} \cdot \overline{RL_4} \cdot \overline{RL_3} \cdot \overline{RL_2} \cdot \overline{RL_1} \cdot \overline{RL_0})$

$V = 0$

Addressing Formats:

See Table A-5.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
IMM	3	3	8E	216	142
DIR	4	2	9E	236	158
EXT	5	3	BE	276	190
IND	6	2	AE	256	174

# LDX

## Load Index Register

Operation:  $IXH \leftarrow (M)$   
 $IXL \leftarrow (M+1)$

Description: Loads the more significant byte of the index register from the byte of memory at the address specified by the program, and loads the less significant byte of the index register from the next byte of memory, at one plus the address specified by the program.

Condition Codes: H: Not affected.  
 I: Not affected.  
 N: Set if the most significant bit of the index register is set by the operation; cleared otherwise.  
 Z: Set if all bits of the index register are cleared by the operation; cleared otherwise.  
 V: Cleared.  
 C: Not affected.

Boolean Formulae for Condition Codes:

$$N = RH_7$$

$$Z = (\overline{RH_7} \cdot \overline{RH_6} \cdot \overline{RH_5} \cdot \overline{RH_4} \cdot \overline{RH_3} \cdot \overline{RH_2} \cdot \overline{RH_1} \cdot \overline{RH_0}) \cdot (\overline{RL_7} \cdot \overline{RL_6} \cdot \overline{RL_5} \cdot \overline{RL_4} \cdot \overline{RL_3} \cdot \overline{RL_2} \cdot \overline{RL_1} \cdot \overline{RL_0})$$

$$V = 0$$

Addressing Formats:

See Table A-5.

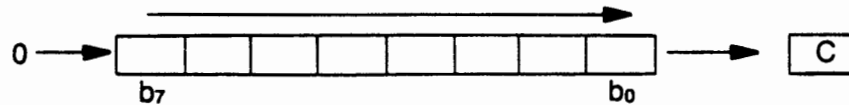
Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
IMM	3	3	CE	316	206
DIR	4	2	DE	336	222
EXT	5	3	FE	376	254
IND	6	2	EE	356	238

## LSR

Logical Shift Right

Operation:



**Description:** Shifts all bits of ACCX or M one place to the right. Bit 7 is loaded with a zero. The C bit is loaded from the least significant bit of ACCX or M.

**Condition Codes:**

- H: Not affected.
- I: Not affected.
- N: Cleared.
- Z: Set if all bits of the result are cleared; cleared otherwise.
- V: Set if, after the completion of the shift operation, EITHER (N is set and C is cleared) OR (N is cleared and C is set); cleared otherwise.
- C: Set if, before the operation, the least significant bit of the ACCX or M was set; cleared otherwise.

**Boolean Formulae for Condition Codes:**

$$N = 0$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$V = N \oplus C = [N \cdot \bar{C}] \vee [\bar{N} \cdot C]$$

(the foregoing formula assumes values of N and C after the shift operation).

$$C = M_0$$

**Addressing Formats:**

See Table A-3.

**Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):**

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A	2	1	44	104	068
B	2	1	54	124	084
EXT	6	3	74	164	116
IND	7	2	64	144	100

**Negate**
**NEG**

Operation:  $ACCX \leftarrow - (ACCX) = 00 - (ACCX)$

or:  $M \leftarrow - (M) = 00 - (M)$

Description: Replaces the contents of ACCX or M with its two's complement. Note that 80 is left unchanged.

Condition Codes: H: Not affected.  
 I: Not affected.  
 N: Set if most significant bit of the result is set; cleared otherwise.  
 Z: Set if all bits of the result are cleared; cleared otherwise.  
 V: Set if there would be two's complement overflow as a result of the implied subtraction from zero; this will occur if and only if the contents of ACCX or M is 80.  
 C: Set if there would be a borrow in the implied subtraction from zero; the C bit will be set in all cases except when the contents of ACCX or M is 00.

Boolean Formulae for Condition Codes:

$$N = R_7$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$V = R_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$C = R_7 + R_6 + R_5 + R_4 + R_3 + R_2 + R_1 + R_0$$

Addressing Formats:

See Table A-3.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A	2	1	40	100	064
B	2	1	50	120	080
EXT	6	3	70	160	112
IND	7	2	60	140	096

## NOP

**No Operation**

**Description:** This is a single-word instruction which causes only the program counter to be incremented. No other registers are affected.

**Condition Codes:** Not affected.

**Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):**

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	2	1	01	001	001



**Inclusive OR**
**ORA**

Operation:  $ACCX \leftarrow (ACCX) \odot (M)$

Description: Perform logical "OR" between the contents of ACCX and the contents of M and places the result in ACCX. (Each bit of ACCX after the operation will be the logical "OR" of the corresponding bits of M and of ACCX before the operation).

Condition Codes: H: Not affected.  
 I: Not affected.  
 N: Set if most significant bit of the result is set; cleared otherwise.  
 Z: Set if all bits of the result are cleared; cleared otherwise.  
 V: Cleared.  
 C: Not affected.

Boolean Formulae for Condition Codes:

$$N = R_7$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$V = 0$$

Addressing Formats:

See Table A-1.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

(DUAL OPERAND)

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A IMM	2	2	8A	212	138
A DIR	3	2	9A	232	154
A EXT	4	3	BA	272	186
A IND	5	2	AA	252	170
B IMM	2	2	CA	312	202
B DIR	3	2	DA	332	218
B EXT	4	3	FA	372	250
B IND	5	2	EA	352	234

## PSH

### Push Data Onto Stack

Operation:  $\downarrow$  (ACCX)  
 $SP \leftarrow (SP) - 0001$

Description: The contents of ACCX is stored in the stack at the address contained in the stack pointer. The stack pointer is then decremented.

Condition Codes: Not affected.

Addressing Formats:

See Table A-4.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A	4	1	36	066	054
B	4	1	37	067	055

**Pull Data from Stack**
**PUL**

Operation:  $SP \leftarrow (SP) + 0001$   
 $\uparrow \text{ACCX}$

Description: The stack pointer is incremented. The ACCX is then loaded from the stack, from the address which is contained in the stack pointer.

Condition Codes: Not affected.

Addressing Formats:

See Table A-4.

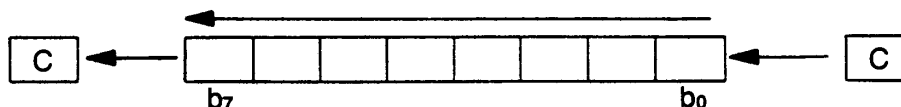
Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A	4	1	32	062	050
B	4	1	33	063	051

## ROL

Rotate Left

Operation:



Description: Shifts all bits of ACCX or M one place to the left. Bit 0 is loaded from the C bit. The C bit is loaded from the most significant bit of ACCX or M.

Condition Codes: H: Not affected.  
 I: Not affected.  
 N: Set if most significant bit of the result is set; cleared otherwise.  
 Z: Set if all bits of the result are cleared; cleared otherwise.  
 V: Set if, after the completion of the operation, EITHER (N is set and C is cleared) OR (N is cleared and C is set); cleared otherwise.  
 C: Set if, before the operation, the most significant bit of the ACCX or M was set; cleared otherwise.

Boolean Formulae for Condition Codes:

$$N = R_7$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$V = N \oplus C = [N \cdot \bar{C}] \odot [\bar{N} \cdot C]$$

(the foregoing formula assumes values of N and C after the rotation)

$$C = M_7$$

Addressing Formats:

See Table A-3

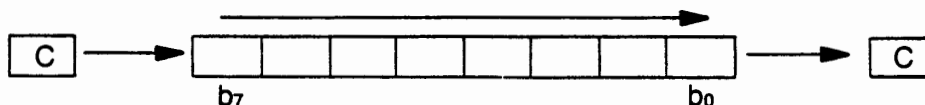
Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A	2	1	49	111	073
B	2	1	59	131	089
EXT	6	3	79	171	121
IND	7	2	69	151	105

# Rotate Right

# ROR

Operation:



Description: Shifts all bits of ACCX or M one place to the right. Bit 7 is loaded from the C bit. The C bit is loaded from the least significant bit of ACCX or M.

Condition Codes:

- H: Not affected.
- I: Not affected.
- N: Set if most significant bit of the result is set; cleared otherwise.
- Z: Set if all bits of the result are cleared; cleared otherwise.
- V: Set if, after the completion of the operation, EITHER (N is set and C is cleared) OR (N is cleared and C is set); cleared otherwise.
- C: Set if, before the operation, the least significant bit of the ACCX or M was set; cleared otherwise.

Boolean Formulae for Condition Codes:

$$N = R_7$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$V = N \oplus C = [N \cdot \bar{C}] \vee [\bar{N} \cdot C]$$

(the foregoing formula assumes values of N and C after the rotation)

$$C = M_0$$

Addressing Formats:

See Table A-3

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A	2	1	46	106	070
B	2	1	56	126	086
EXT	6	3	76	166	118
IND	7	2	66	146	102

# RTI

## Return from Interrupt

Operation:

$$SP \leftarrow (SP) + 0001, \uparrow CC$$

$$SP \leftarrow (SP) + 0001, \uparrow ACCB$$

$$SP \leftarrow (SP) + 0001, \uparrow ACCA$$

$$SP \leftarrow (SP) + 0001, \uparrow IXH$$

$$SP \leftarrow (SP) + 0001, \uparrow IXL$$

$$SP \leftarrow (SP) + 0001, \uparrow PCH$$

$$SP \leftarrow (SP) + 0001, \uparrow PCL$$

Description: The condition codes, accumulators B and A, the index register, and the program counter, will be restored to a state pulled from the stack. Note that the interrupt mask bit will be reset if and only if the corresponding bit stored in the stack is zero.

Condition Codes: Restored to the states pulled from the stack.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	10	1	3B	073	059

## Return from Interrupt

### Example

		Memory Location	Machine Code (Hex)	Label	Assembler Language Operator	Operand
A. Before	PC	→ \$D066	3B		RTI	
	SP	→ \$EFF8				
		\$EFF9	11HINZVC	(binary)		
		\$EFFA	12			
		\$EFFB	34			
		\$EFFC	56			
		\$EFFD	78			
		\$EFFE	55			
		\$EFFF	67			
B. After	PC	→ \$5567	**		***	*****
		\$EFF8				
		\$EFF9	11HINZVC	(binary)		
		\$EFFA	12			
		\$EFFB	34			
		\$EFFC	56			
		\$EFFD	78			
		\$EFFE	55			
	SP	→ \$EFFF	67			

CC = HINZVC (binary)

ACCB = 12 (Hex)

IXH = 56 (Hex)

ACCA = 34 (Hex)

IXL = 78 (Hex)

**Return from Subroutine**
**RTS**

Operation:  $SP \leftarrow (SP) + 0001$   
 $\uparrow PCH$   
 $SP \leftarrow (SP) + 0001$   
 $\uparrow PCL$

Description: The stack pointer is incremented (by 1). The contents of the byte of memory, at the address now contained in the stack pointer, are loaded into the 8 bits of highest significance in the program counter. The stack pointer is again incremented (by 1). The contents of the byte of memory, at the address now contained in the stack pointer, are loaded into the 8 bits of lowest significance in the program counter.

Condition Codes: Not affected.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	5	1	39	071	057

**Return from Subroutine**
**EXAMPLE**

	Memory Location	Machine Code (Hex)	Label	Assembler Language Operator	Operand
A. <i>Before</i>					
PC	\$30A2	39		RTS	
SP	\$EFFF				
	\$EFFF	02			
B. <i>After</i>					
PC	\$1002	**		***	*****
	\$EFFF				
	\$EFFF	10			
SP	\$EFFF	02			

## SBA

### Subtract Accumulators

Operation:  $ACCA \leftarrow (ACCA) - (ACCB)$

Description: Subtracts the contents of ACCB from the contents of ACCA and places the result in ACCA. The contents of ACCB are not affected.

Condition Codes: H: Not affected.  
 I: Not affected.  
 N: Set if most significant bit of the result is set; cleared otherwise.  
 Z: Set if all bits of the result are cleared; cleared otherwise.  
 V: Set if there was two's complement overflow as a result of the operation.  
 C: Carry is set if the absolute value of accumulator B plus previous carry is larger than the absolute value of accumulator A; reset otherwise.

Boolean Formulae for Condition Codes:

$$N = R_7$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$V = A_7 \cdot \bar{B}_7 \cdot \bar{R}_7 + \bar{A}_7 \cdot B_7 \cdot R_7$$

$$C = \bar{A}_7 \cdot B_7 + B_7 \cdot R_7 + R_7 \cdot \bar{A}_7$$

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	2	1	10	020	016



**Subtract with Carry**
**SBC**

Operation:  $ACCX \leftarrow (ACCX) - (M) - (C)$

Description: Subtracts the contents of M and C from the contents of ACCX and places the result in ACCX.

Condition Codes: H: Not affected.  
 I: Not affected.  
 N: Set if most significant bit of the result is set; cleared otherwise.  
 Z: Set if all bits of the result are cleared; cleared otherwise.  
 V: Set if there was two's complement overflow as a result of the operation; cleared otherwise.  
 C: Carry is set if the absolute value of the contents of memory plus previous carry is larger than the absolute value of the accumulator; reset otherwise.

Boolean Formulae for Condition Codes:

$$N = R_7$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$V = X_7 \cdot \bar{M}_7 \cdot \bar{R}_7 + \bar{X}_7 \cdot M_7 \cdot R_7$$

$$C = \bar{X}_7 \cdot M_7 + M_7 \cdot R_7 + R_7 \cdot \bar{X}_7$$

Addressing Formats:

See Table A-1.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

(DUAL OPERAND)

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A IMM	2	2	82	202	130
A DIR	3	2	92	222	146
A EXT	4	3	B2	262	178
A IND	5	2	A2	242	162
B IMM	2	2	C2	302	194
B DIR	3	2	D2	322	210
B EXT	4	3	F2	362	242
B IND	5	2	E2	342	226

## SEC

**Set Carry**

 Operation: C bit  $\leftarrow$  1

Description: Sets the carry bit in the processor condition codes register.

 Condition Codes: H: Not affected.  
 I: Not affected.  
 N: Not affected.  
 Z: Not affected.  
 V: Not affected.  
 C: Set.

Boolean Formulae for Condition Codes:

$$C = 1$$

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	2	1	0D	015	013

**Set Interrupt Mask**
**SEI**

Operation: I bit  $\leftarrow$  1

Description: Sets the interrupt mask bit in the processor condition codes register. The microprocessor is inhibited from servicing an interrupt from a peripheral device, and will continue with execution of the instructions of the program, until the interrupt mask bit has been cleared.

Condition Codes: H: Not affected.  
 I: Set.  
 N: Not affected.  
 Z: Not affected.  
 V: Not affected.  
 C: Not affected.

Boolean Formulae for Condition Codes:  
 I = 1

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	2	1	0F	017	015

## SEV

### Set Two's Complement Overflow Bit

Operation:  $V \text{ bit} \leftarrow 1$

Description: Sets the two's complement overflow bit in the processor condition codes register.

Condition Codes: H: Not affected.  
 I: Not affected.  
 N: Not affected.  
 Z: Not affected.  
 V: Set.  
 C: Not affected.

Boolean Formulae for Condition Codes:

$$V = 1$$

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	2	1	0B	013	011

# STA

## Store Accumulator

Operation:  $M \leftarrow (ACCX)$

Description: Stores the contents of ACCX in memory. The contents of ACCX remains unchanged.

Condition Codes: H: Not affected.  
 I: Not affected.  
 N: Set if the most significant bit of the contents of ACCX is set; cleared otherwise.  
 Z: Set if all bits of the contents of ACCX are cleared; cleared otherwise.  
 V: Cleared.  
 C: Not affected.

Boolean Formulae for Condition Codes:

$$N = X_7$$

$$Z = \overline{X_7} \cdot \overline{X_6} \cdot \overline{X_5} \cdot \overline{X_4} \cdot \overline{X_3} \cdot \overline{X_2} \cdot \overline{X_1} \cdot \overline{X_0}$$

$$V = 0$$

Addressing Formats:

See Table A-2.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A DIR	4	2	97	227	151
A EXT	5	3	B7	267	183
A IND	6	2	A7	247	167
B DIR	4	2	D7	327	215
B EXT	5	3	F7	367	247
B IND	6	2	E7	347	231

# STS

## Store Stack Pointer

Operation:  $M \leftarrow (SPH)$   
 $M + 1 \leftarrow (SPL)$

Description: Stores the more significant byte of the stack pointer in memory at the address specified by the program, and stores the less significant byte of the stack pointer at the next location in memory, at one plus the address specified by the program.

Condition Codes: H: Not affected.  
 I: Not affected.  
 N: Set if the most significant bit of the stack pointer is set; cleared otherwise.  
 Z: Set if all bits of the stack pointer are cleared; cleared otherwise.  
 V: Cleared.  
 C: Not affected.

Boolean Formulae for Condition Codes:

$$N = SPH_7$$

$$Z = (\overline{SPH_7} \cdot \overline{SPH_6} \cdot \overline{SPH_5} \cdot \overline{SPH_4} \cdot \overline{SPH_3} \cdot \overline{SPH_2} \cdot \overline{SPH_1} \cdot \overline{SPH_0}) \cdot (\overline{SPL_7} \cdot \overline{SPL_6} \cdot \overline{SPL_5} \cdot \overline{SPL_4} \cdot \overline{SPL_3} \cdot \overline{SPL_2} \cdot \overline{SPL_1} \cdot \overline{SPL_0})$$

$$V = 0$$

Addressing Formats:

See Table A-6.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
DIR	5	2	9F	237	159
EXT	6	3	BF	277	191
IND	7	2	AF	257	175

**Store Index Register**
**STX**

Operation:  $M \leftarrow (IXH)$   
 $M + 1 \leftarrow (IXL)$

Description: Stores the more significant byte of the index register in memory at the address specified by the program, and stores the less significant byte of the index register at the next location in memory, at one plus the address specified by the program.

Condition Codes: H: Not affected.  
 I: Not affected.  
 N: Set if the most significant bite of the index register is set; cleared otherwise.  
 Z: Set if all bits of the index register are cleared; cleared otherwise.  
 V: Cleared.  
 C: Not affected.

Boolean Formulae for Condition Codes:

$$N = IXH_7$$

$$Z = (\overline{IXH_7} \cdot \overline{IXH_6} \cdot \overline{IXH_5} \cdot \overline{IXH_4} \cdot \overline{IXH_3} \cdot \overline{IXH_2} \cdot \overline{IXH_1} \cdot \overline{IXH_0}) \cdot (\overline{IXL_7} \cdot \overline{IXL_6} \cdot \overline{IXL_5} \cdot \overline{IXL_4} \cdot \overline{IXL_3} \cdot \overline{IXL_2} \cdot \overline{IXL_1} \cdot \overline{IXL_0})$$

$$V = 0$$

Addressing Formats:

See Table A-6.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
DIR	5	2	DF	337	223
EXT	6	3	FF	377	255
IND	7	2	EF	357	239

# SUB

**Subtract**

Operation:  $ACCX \leftarrow (ACCX) - (M)$

Description: Subtracts the contents of M from the contents of ACCX and places the result in ACCX.

Condition Codes: H: Not affected.  
 I: Not affected.  
 N: Set if most significant bit of the result is set; cleared otherwise.  
 Z: Set if all bits of the result are cleared; cleared otherwise.  
 V: Set if there was two's complement overflow as a result of the operation; cleared otherwise.  
 C: Set if the absolute value of the contents of memory are larger than the absolute value of the accumulator; reset otherwise.

Boolean Formulae for Condition Codes:

$$N = R_7$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$V = X_7 \cdot \bar{M}_7 \cdot \bar{R}_7 \cdot \bar{X}_7 \cdot M_7 \cdot R_7$$

$$C = \bar{X}_7 \cdot M_7 + M_7 \cdot R_7 + R_7 \cdot \bar{X}_7$$

Addressing Formats:

See Table A-1.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

(DUAL OPERAND)

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A IMM	2	2	80	200	128
A DIR	3	2	90	220	144
A EXT	4	3	B0	260	176
A IND	5	2	A0	240	160
B IMM	2	2	C0	300	192
B DIR	3	2	D0	320	208
B EXT	4	3	F0	360	240
B IND	5	2	E0	340	224



**Software Interrupt**
**SWI**

Operation:

$$PC \leftarrow (PC) + 0001$$

$$\downarrow (PCL), SP \leftarrow (SP) - 0001$$

$$\downarrow (PCH), SP \leftarrow (SP) - 0001$$

$$\downarrow (IXL), SP \leftarrow (SP) - 0001$$

$$\downarrow (IXH), SP \leftarrow (SP) - 0001$$

$$\downarrow (ACCA), SP \leftarrow (SP) - 0001$$

$$\downarrow (ACCB), SP \leftarrow (SP) - 0001$$

$$\downarrow (CC), SP \leftarrow (SP) - 0001$$

$$I \leftarrow 1$$

$$PCH \leftarrow (n-0005)$$

$$PCL \leftarrow (n-0004)$$

Description: The program counter is incremented (by 1). The program counter, index register, and accumulator A and B, are pushed into the stack. The condition codes register is then pushed into the stack, with condition codes H, I, N, Z, V, C going respectively into bit positions 5 thru 0, and the top two bits (in bit positions 7 and 6) are set (to the 1 state). The stack pointer is decremented (by 1) after each byte of data is stored in the stack.

The interrupt mask bit is then set. The program counter is then loaded with the address stored in the software interrupt pointer at memory locations (n-5) and (n-4), where n is the address corresponding to a high state on all lines of the address bus.

Condition Codes:

H: Not affected.

I: Set.

N: Not affected.

Z: Not affected.

V: Not affected.

C: Not affected.

Boolean Formula for Condition Codes:

$$I = 1$$

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	12	1	3F	077	063

**Software Interrupt**
**EXAMPLE**
**A. Before:**

CC = HINZVC (binary)

ACCB = 12 (Hex)

ACCA = 34 (Hex)

IXH = 56 (Hex)

IXL = 78 (Hex)

		Memory	Machine	Assembler Language		
		Location	Code (Hex)	Label	Operator	Operand
PC	→	\$5566	3F		SWI	
SP	→	\$EFFF				
		\$FFFA	D0			
		\$FFFB	55			

**B. After:**

PC → \$D055

SP → \$EFF8

\$EFF9 11HINZVC (binary)

\$EFFA 12

\$EFFB 34

\$EFFC 56

\$EFFD 78

\$EFFE 55

\$EFFF 67

Note: This example assumes that FFFF is the memory location addressed when all lines of the address bus go to the high state.

# TAB

## Transfer from Accumulator A to Accumulator B

Operation:  $ACCB \leftarrow (ACCA)$

Description: Moves the contents of ACCA to ACCB. The former contents of ACCB are lost. The contents of ACCA are not affected.

Condition Codes: H: Not affected.  
 I: Not affected.  
 N: Set if the most significant bit of the contents of the accumulator is set; cleared otherwise.  
 Z: Set if all bits of the contents of the accumulator are cleared; cleared otherwise.  
 V: Cleared.  
 C: Not affected.

Boolean Formulae for Condition Codes:

$$N = R_7$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$V = 0$$

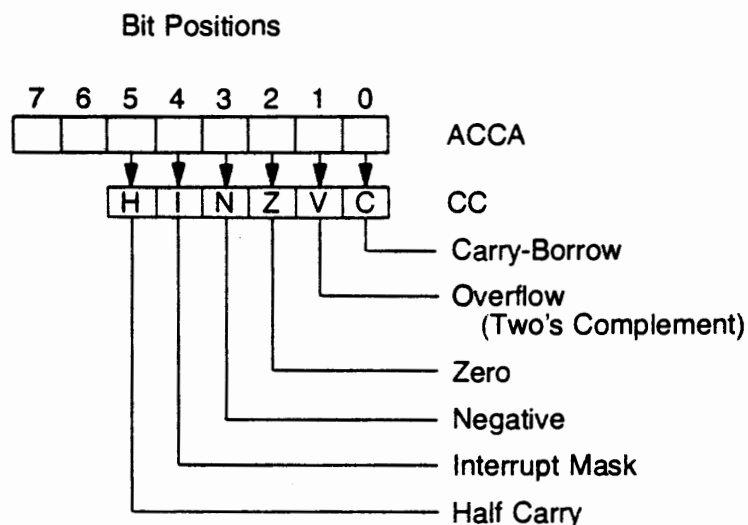
Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	2	1	16	026	022

# TAP

Transfer from Accumulator A  
to Processor Condition Codes Register

Operation:  $CC \leftarrow (ACCA)$



**Description:** Transfers the contents of bit positions 0 thru 5 of accumulator A to the corresponding bit positions of the processor condition codes register. The contents of accumulator A remain unchanged.

**Condition Codes:** Set or reset according to the contents of the respective bits 0 thru 5 of accumulator A.

**Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):**

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	2	1	06	006	006

**Transfer from Accumulator B to Accumulator A**
**TBA**

 Operation:  $ACCA \leftarrow (ACCB)$ 

Description: Moves the contents of ACCB to ACCA. The former contents of ACCA are lost. The contents of ACCB are not affected.

 Condition Codes: H: Not affected.  
 I: Not affected.  
 N: Set if the most significant accumulator bit is set; cleared otherwise.  
 Z: Set if all accumulator bits are cleared; cleared otherwise.  
 V: Cleared.  
 C: Not affected.

Boolean Formulae for Condition Codes:

$$N = R_7$$

$$Z = \bar{R}_7 \cdot \bar{R}_6 \cdot \bar{R}_5 \cdot \bar{R}_4 \cdot \bar{R}_3 \cdot \bar{R}_2 \cdot \bar{R}_1 \cdot \bar{R}_0$$

$$V = 0$$

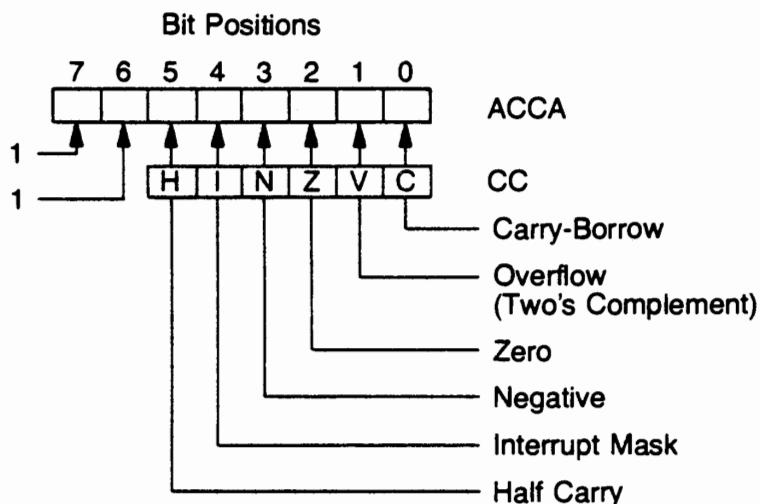
Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	2	1	17	027	023

# TPA

## Transfer from Processor Condition Codes Register to Accumulator A

Operation:  $ACCA \leftarrow (CC)$



**Description:** Transfers the contents of the processor condition codes register to corresponding bit positions 0 thru 5 of accumulator A. Bit positions 6 and 7 of accumulator A are set (i.e. go to the "1" state). The processor condition codes register remains unchanged.

**Condition Codes:** Not affected.

**Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):**

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	2	1	07	007	007

**Test**
**TST**

Operation: (ACCX) – 00  
 (M) – 00

Description: Set condition codes N and Z according to the contents of ACCX or M.

Condition Codes: H: Not affected.  
 I: Not affected.  
 N: Set if most significant bit of the contents of ACCX or M is set; cleared otherwise.  
 Z: Set if all bits of the contents of ACCX or M are cleared; cleared otherwise.  
 V: Cleared.  
 C: Cleared.

Boolean Formulae for Condition Codes:

$$N = M_7$$

$$Z = \bar{M}_7 \cdot \bar{M}_6 \cdot \bar{M}_5 \cdot \bar{M}_4 \cdot \bar{M}_3 \cdot \bar{M}_2 \cdot \bar{M}_1 \cdot \bar{M}_0$$

$$V = 0$$

$$C = 0$$

Addressing Formats:

See Table A-3.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
A	2	1	4D	115	077
B	2	1	5D	135	093
EXT	6	3	7D	175	125
IND	7	2	6D	155	109

**TSX****Transfer from Stack Pointer to Index Register**

Operation:  $IX \leftarrow (SP) + 0001$

Description: Loads the index register with one plus the contents of the stack pointer. The contents of the stack pointer remain unchanged.

Condition Codes: Not affected.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	4	1	30	060	048



**Transfer From Index Register to Stack Pointer**
**TXS**

 Operation:  $SP \leftarrow (IX) - 0001$ 

 Description: Loads the stack pointer with the contents of the index register, minus one.  
 The contents of the index register remain unchanged.

Condition Codes: Not affected.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	4	1	35	065	053

# WAI

## Wait for Interrupt

Operation:  $PC \leftarrow (PC) + 0001$   
 $\downarrow (PCL), SP \leftarrow (SP) - 0001$   
 $\downarrow (PCH), SP \leftarrow (SP) - 0001$   
 $\downarrow (IXL), SP \leftarrow (SP) - 0001$   
 $\downarrow (IXH), SP \leftarrow (SP) - 0001$   
 $\downarrow (ACCA), SP \leftarrow (SP) - 0001$   
 $\downarrow (ACCB), SP \leftarrow (SP) - 0001$   
 $\downarrow (CC), SP \leftarrow (SP) - 0001$

Condition Codes: Not affected.

Description: The program counter is incremented (by 1). The program counter, index register, and accumulators A and B, are pushed into the stack. The condition codes register is then pushed into the stack, with condition codes H, I, N, Z, V, C going respectively into bit positions 5 thru 0, and the top two bits (in bit positions 7 and 6) are set (to the 1 state). The stack pointer is decremented (by 1) after each byte of data is stored in the stack.

Execution of the program is then suspended until an interrupt from a peripheral device is signalled, by the interrupt request control input going to a low state.

When an interrupt is signalled on the interrupt request line, and provided the I bit is clear, execution proceeds as follows. The interrupt mask bit is set. The program counter is then loaded with the address stored in the internal interrupt pointer at memory locations (n-7) and (n-6), where n is the address corresponding to a high state on all lines of the address bus.

Condition Codes: H: Not affected.  
 I: Not affected until an interrupt request signal is detected on the interrupt request control line. When the interrupt request is received the I bit is set and further execution takes place, provided the I bit was initially clear.  
 N: Not affected.  
 Z: Not affected.  
 V: Not affected.  
 C: Not affected.

Addressing Modes, Execution Time, and Machine Code (hexadecimal/ octal/ decimal):

Addressing Modes	Execution Time (No. of cycles)	Number of bytes of machine code	Coding of First (or only) byte of machine code		
			HEX.	OCT.	DEC.
INHERENT	9	1	3E	076	062

Addressing Mode of Second Operand	First Operand	
	Accumulator A	Accumulator B
IMMediate	CCC A #number CCC A #symbol CCC A #expression CCC A #'C	CCC B #number CCC B #symbol CCC B #expression CCC B #'C
DIRect or EXTended	CCC A number CCC A symbol CCC A expression	CCC B number CCC B symbol CCC B expression
INDexed	CCC A X CCC Z ,X CCC A number,X CCC A symbol,X CCC A expression,X	CCC B X CCC B ,X CCC B number,X CCC B symbol,X CCC B expression,X

- Notes: 1. CCC = mnemonic operator of source instruction.  
 2. "symbol" may be the special symbol "\*".  
 3. "expression" may contain the special symbol "\*".  
 4. space may be omitted before A or B.

Applicable to the following source instructions:

ADC ADD AND BIT CMP  
 EOR LDA ORA SBC SUB

\*Special symbol indicating program-counter.

**TABLE A-1. Addressing Formats (1)**

Addressing Mode of Second Operand	First Operand	
	Accumulator A	Accumulator B
DIRect or EXTended	STA A number STA A symbol STA A expression	STA B number STA B symbol STA B expression
INDexed	STA A X STA A ,X STA A number,X STA A symbol,X STA A expression,X	STA B X STA B ,X STA B number,X STA B symbol,X STA B expression,X

- Notes: 1. "symbol" may be the special symbol "\*".  
 2. "expression" may contain the special symbol "\*".  
 3. Space may be omitted before A or B.

Applicable to the source instruction:

STA

\*Special symbol indicating program-counter.

**TABLE A-2. Addressing Formats (2)**

Operand or Addressing Mode	Formats
Accumulator A	CCC A
Accumulator B	CCC B
EXTended	CCC number CCC symbol CCC expression
INDexed	CCC X CCC ,X CCC number,X CCC symbol,X CCC expression,X

**Notes:** 1. CCC = mnemonic operator of source instruction.  
 2. "symbol" may be the special symbol "\*".  
 3. "expression" may contain the special symbol "\*".  
 4. Space may be omitted before A or B.

Applicable to the following source instructions:

ASL ASR CLR COM DEC INC  
 LSR NEG ROL ROR TST

\*Special symbol indicating program-counter.

**TABLE A-3. Addressing Formats (3)**

---

Operand	Formats
Accumulator A	CCC A
Accumulator B	CCC B

**Notes:** 1. CCC = mnemonic operator of source instruction.  
 2. Space may be omitted before A or B.

Applicable to the following source instructions:

PSH PUL

**TABLE A-4. Addressing Formats (4)**

Addressing Mode	Formats
IMMediate	CCC #number CCC #symbol CCC #expression CCC #'C
DIRect or EXTended	CCC number CCC symbol CCC expression
INDexed	CCC X CCC ,X CCC number,X CCC symbol,X CCC expression,X

- Notes: 1. CCC = mnemonic operator of source instruction.  
 2. "symbol" may be the special symbol "\*".  
 3. "expression" may contain the special symbol "\*".

Applicable to the following source instructions:

CPX LDS LDX

\*Special symbol indicating program-counter.

**TABLE A-5. Addressing Formats (5)**

Addressing Mode	Formats
DIRect or EXTended	CCC number CCC symbol CCC expression
INDexed	CCC X CCC ,X CCC number,X CCC symbol,X CCC expression,X

- Notes: 1. CCC = mnemonic operator of source instruction.  
 2. "symbol" may be the special symbol "\*".  
 3. "expression" may contain the special symbol "\*".

Applicable to the following source instructions:

STS STX

\*Special symbol indicating program-counter.

**TABLE A-6. Addressing Formats (6)**

Addressing Mode	Formats
EXTended	CCC number CCC symbol CCC expression
INDexed	CCC X CCC ,X CCC number,X CCC symbol,X CCC expression,X

**Notes:** 1. CCC = mnemonic operator of source instruction.  
 2. "symbol" may be the special symbol "\*\*".  
 3. "expression" may contain the special symbol "\*\*".

Applicable to the following source instructions:

JMP JSR

\*Special symbol indicating program-counter.

**TABLE A-7. Addressing Formats (7)**

Addressing Mode	Formats
RELative	CCC number CCC symbol CCC expression

**Notes:** 1. CCC = mnemonic operator of source instruction.  
 2. "symbol" may be the special symbol "\*\*".  
 3. "expression" may contain the special symbol "\*\*".

Applicable to the following source instructions:

BCC BCS BEQ BGE BGT BHI BLE BLS  
 BLT BMI BNE BPL BRA BSR BVC BVS

\*Special symbol indicating program-counter.

**TABLE A-8. Addressing Formats (8)**



# Individual Learning Program

## MICROPROCESSORS

### *Appendix B* DATA SHEETS

EE-3401

Courtesy of  
Motorola Semiconductor  
Products, Inc

## CONTENTS

MC6800 Data Sheet .....	Page B-3
MC6820 Data Sheet .....	Page B-21
MC6850 Data Sheet .....	Page B-31
MCM6810A Data Sheet .....	Page B-39
MCM6830A Data Sheet .....	Page B-43
MCM6832 Data Sheet .....	Page B-47
Positive Powers of two .....	Page B-51
Negative Powers of two .....	Page B-52
Positive Powers of eight .....	Page B-53
Positive Powers of sixteen .....	Page B-53
Negative Powers of sixteen .....	Page B-53




**MOTOROLA**  
**Semiconductors**

3501 ED BLUESTEIN BLVD. AUSTIN, TEXAS 78721

## Advance Information

### MICROPROCESSOR WITH CLOCK

The MC6808 is a monolithic 8-bit microprocessor that contains all the registers and accumulators of the present MC6800 plus an internal clock oscillator and driver on the same chip.

The MC6808 is completely software-compatible with the MC6800 as well as the entire M6800 family of parts. Hence the MC6808 is expandable to 65K words.

This very cost-effective MPU allows the designer to use the MC6808 in consumer as well as industrial applications without sacrificing industrial specifications.

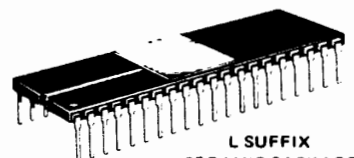
- On-Chip Clock Circuit
- Software-Compatible with the MC6800
- Expandable to 65K words
- Standard TTL-Compatible Inputs and Outputs
- 8-Bit Word Size
- 16-Bit Memory Addressing
- Interrupt Capability

# MC6808

## MOS

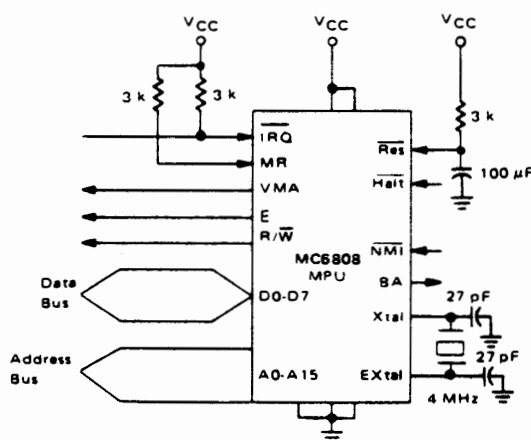
 (N-CHANNEL, SILICON-GATE,  
 DEPLETION LOAD)

### MICROPROCESSOR WITH CLOCK


 L SUFFIX  
 CERAMIC PACKAGE  
 CASE 715

 P SUFFIX  
 PLASTIC PACKAGE  
 CASE 711

FIGURE 1 – TYPICAL MICROPROCESSOR INTERFACE



PIN ASSIGNMENT

1	VSS	Reset	40
2	Halt	EXTal	39
3	MR	Xtal	38
4	IRQ	E	37
5	VMA	VSS	36
6	NMI	VCC	35
7	BA	R/W	34
8	VCC	D0	33
9	A0	D1	32
10	A1	D2	31
11	A2	D3	30
12	A3	D4	29
13	A4	D5	28
14	A5	D6	27
15	A6	D7	26
16	A7	A15	25
17	A8	A14	24
18	A9	A13	23
19	A10	A12	22
20	A11	VSS	21





**MOTOROLA**  
**Semiconductors**

BOX 20912 • PHOENIX, ARIZONA 85036

**MC6800**

(0 to 70°C; L or P Suffix)

**MC6800C**

(-40 to 85°C; L Suffix only)

### MICROPROCESSING UNIT (MPU)

The MC6800 is a monolithic 8-bit microprocessor forming the central control function for Motorola's M6800 family. Compatible with TTL, the MC6800, as with all M6800 system parts, requires only one +5.0-volt power supply, and no external TTL devices for bus interface.

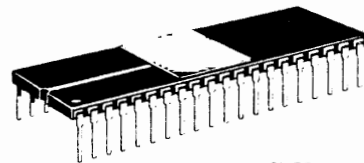
The MC6800 is capable of addressing 65K bytes of memory with its 16-bit address lines. The 8-bit data bus is bidirectional as well as 3-state, making direct memory addressing and multiprocessor applications realizable.

- Eight-Bit Parallel Processing
- Bi-Directional Data Bus
- Sixteen-Bit Address Bus — 65K Bytes of Addressing
- 72 Instructions — Variable Length
- Seven Addressing Modes — Direct, Relative, Immediate, Indexed, Extended, Implied and Accumulator
- Variable Length Stack
- Vectored Restart
- Maskable Interrupt Vector
- Separate Non-Maskable Interrupt — Internal Registers Saved In Stack
- Six Internal Registers — Two Accumulators, Index Register, Program Counter, Stack Pointer and Condition Code Register
- Direct Memory Addressing (DMA) and Multiple Processor Capability
- Clock Rates as High as 1 MHz
- Simple Bus Interface Without TTL
- Halt and Single Instruction Execution Capability

**MOS**

(N-CHANNEL, SILICON-GATE)

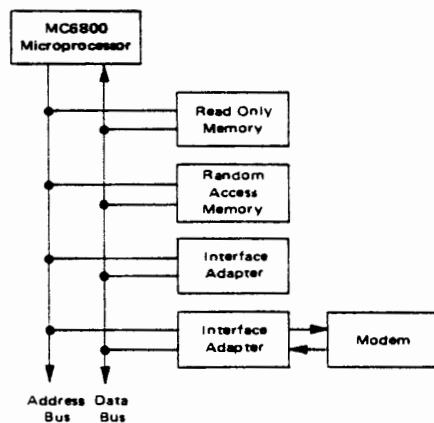
**MICROPROCESSOR**



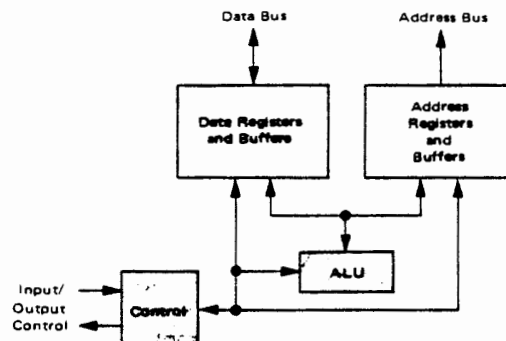
**L SUFFIX**  
CERAMIC PACKAGE  
CASE 715

NOT SHOWN: P SUFFIX  
PLASTIC PACKAGE  
CASE 711

**M6800 MICROCOMPUTER FAMILY  
BLOCK DIAGRAM**



**MC6800 MICROPROCESSOR  
BLOCK DIAGRAM**



## MC6800

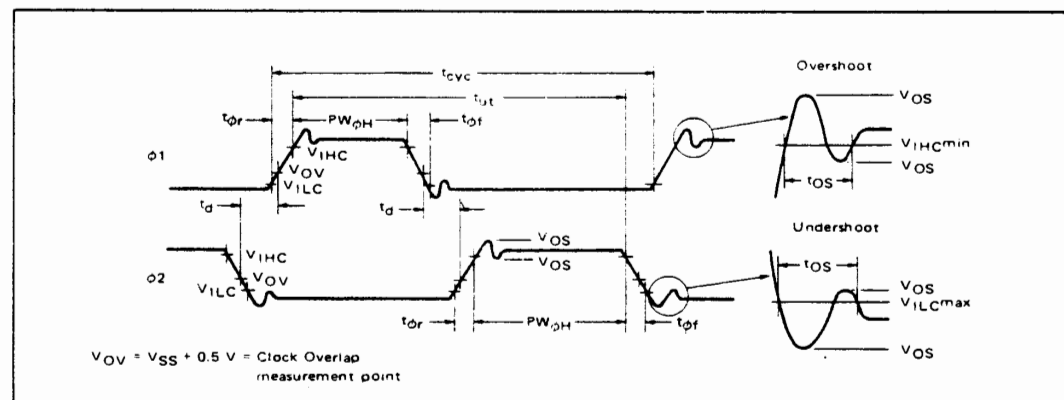
ELECTRICAL CHARACTERISTICS ( $V_{CC} = 5.0 \text{ V} \pm 5\%$ ,  $V_{SS} = 0$ ,  $T_A = 0 \text{ to } 70^\circ\text{C}$  unless otherwise noted.)

Characteristic		Symbol	Min	Typ	Max	Unit
Input High Voltage	Logic $\phi 1, \phi 2$	$V_{IH}$ $V_{IHC}$	$V_{SS} + 2.0$ $V_{CC} - 0.3$	—	$V_{CC}$ $V_{CC} + 0.1$	Vdc
Input Low Voltage	Logic $\phi 1, \phi 2$	$V_{IL}$ $V_{ILC}$	$V_{SS} - 0.3$ $V_{SS} - 0.1$	—	$V_{SS} + 0.8$ $V_{SS} + 0.3$	Vdc
Clock Overshoot/Undershoot — Input High Level — Input Low Level		$V_{OS}$	$V_{CC} - 0.5$ $V_{SS} - 0.5$	—	$V_{CC} + 0.5$ $V_{SS} + 0.5$	Vdc
Input Leakage Current ( $V_{in} = 0 \text{ to } 5.25 \text{ V}$ , $V_{CC} = \text{max}$ ) ( $V_{in} = 0 \text{ to } 5.25 \text{ V}$ , $V_{CC} = 0.0 \text{ V}$ )	Logic* $\phi 1, \phi 2$	$I_{in}$	— —	1.0 —	2.5 100	$\mu\text{Adc}$
Three-State (Off State) Input Current ( $V_{in} = 0.4 \text{ to } 2.4 \text{ V}$ , $V_{CC} = \text{max}$ )	D0-D7 A0-A15, R/W	$I_{TSI}$	— —	2.0 —	10 100	$\mu\text{Adc}$
Output High Voltage ( $I_{Load} = -205 \mu\text{Adc}$ , $V_{CC} = \text{min}$ ) ( $I_{Load} = -145 \mu\text{Adc}$ , $V_{CC} = \text{min}$ ) ( $I_{Load} = -100 \mu\text{Adc}$ , $V_{CC} = \text{min}$ )	D0-D7 A0-A15, R/W, VMA BA	$V_{OH}$	$V_{SS} + 2.4$ $V_{SS} + 2.4$ $V_{SS} + 2.4$	— — —	— — —	Vdc
Output Low Voltage ( $I_{Load} = 1.6 \text{ mA}$ , $V_{CC} = \text{min}$ )		$V_{OL}$	—	—	$V_{SS} + 0.4$	Vdc
Power Dissipation		$P_D$	—	0.600	1.2	W
Capacitance * ( $V_{in} = 0$ , $T_A = 25^\circ\text{C}$ , $f = 1.0 \text{ MHz}$ )	$\phi 1, \phi 2$ TSC DBE D0-D7 Logic Inputs A0-A15, R/W, VMA	$C_{in}$     $C_{out}$	80 — — — — —	120 — 7.0 10 6.5 —	160 15 10 12.5 8.5 12	pF
Frequency of Operation		$f$	0.1	—	1.0	MHz
Clock Timing (Figure 1) Cycle Time		$t_{cyc}$	1.0	—	10	$\mu\text{s}$
Clock Pulse Width (Measured at $V_{CC} - 0.3 \text{ V}$ )	$\phi 1$ $\phi 2$	$PW_{\phi H}$	430 450	— —	4500 4500	ns
Total $\phi 1$ and $\phi 2$ Up Time		$t_{ut}$	940	—	—	ns
Rise and Fall Times (Measured between $V_{SS} + 0.3 \text{ V}$ and $V_{CC} - 0.3 \text{ V}$ )	$\phi 1, \phi 2$	$t_{pr}, t_{of}$	5.0	—	50	ns
Delay Time or Clock Separation (Measured at $V_{OV} = V_{SS} + 0.5 \text{ V}$ )		$t_d$	0	—	9100	ns
Overshoot Duration		$t_{OS}$	0	—	40	ns

\*Except  $\overline{IRQ}$  and  $\overline{NMI}$ , which require  $3 \text{ k}\Omega$  pullup load resistors for wire-OR capability at optimum operation.

\*Capacitances are periodically sampled rather than 100% tested.

FIGURE 1 — CLOCK TIMING WAVEFORM



**MC6800**
**MAXIMUM RATINGS**

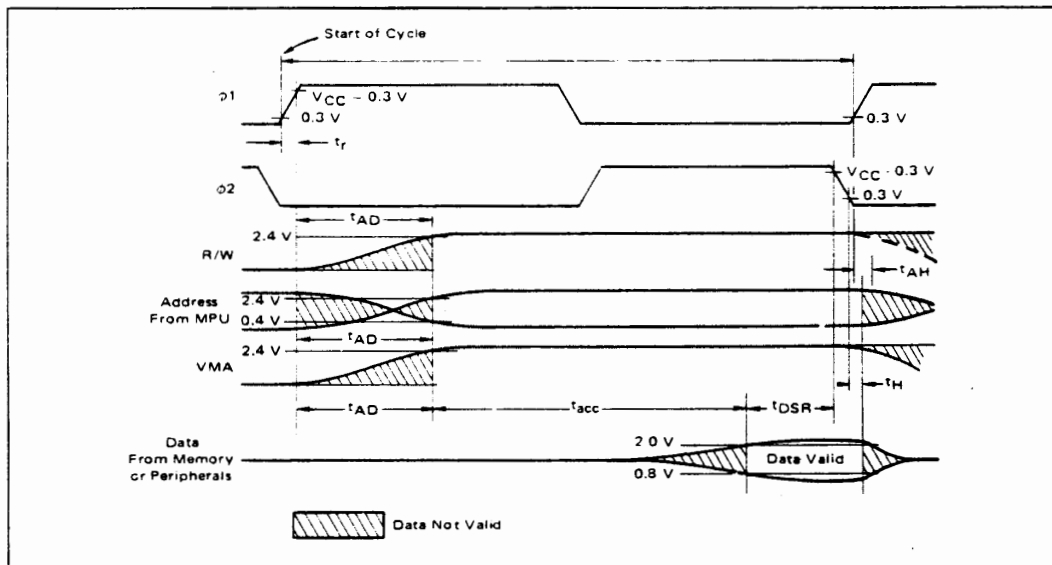
Rating	Symbol	Value	Unit
Supply Voltage	$V_{CC}$	-0.3 to +7.0	Vdc
Input Voltage	$V_{in}$	-0.3 to +7.0	Vdc
Operating Temperature Range	$T_A$	0 to +70	°C
Storage Temperature Range	$T_{stg}$	-55 to +150	°C
Thermal Resistance	$\theta_{JA}$	70	°C/W

This device contains circuitry to protect the inputs against damage due to high static voltages or electric fields; however, it is advised that normal precautions be taken to avoid application of any voltage higher than maximum rated voltages to this high impedance circuit.

**READ/WRITE TIMING** Figures 2 and 3,  $f = 1.0$  MHz, Load Circuit of Figure 6.

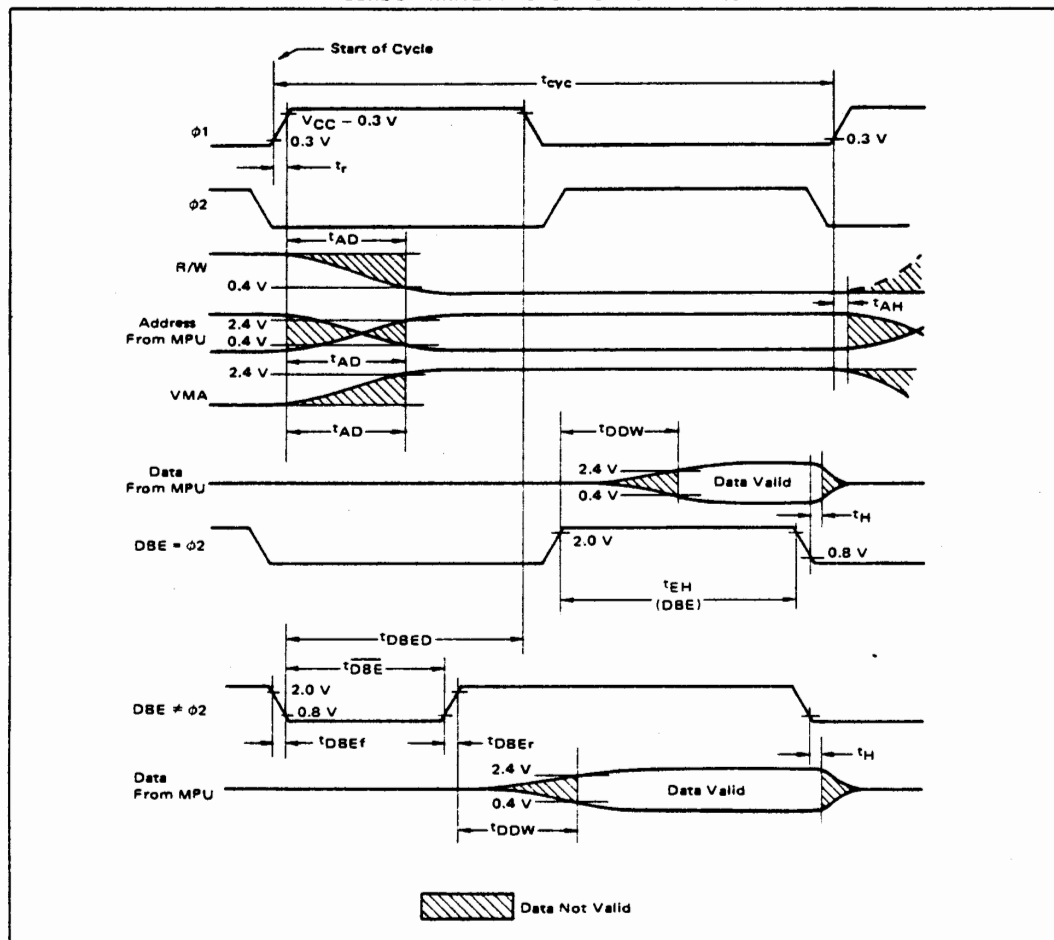
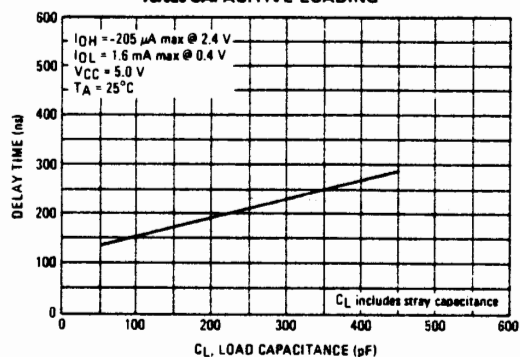
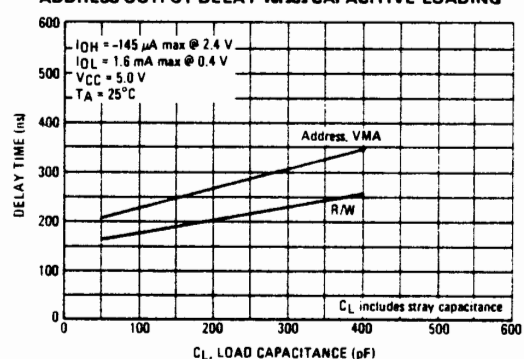
Characteristic	Symbol	Min	Typ	Max	Unit
Address Delay	$t_{AD}$	—	220	300	ns
Peripheral Read Access Time $t_{acc} = t_{ut} - (t_{AD} + t_{DSR})$	$t_{acc}$	—	—	540	ns
Data Setup Time (Read)	$t_{DSR}$	100	—	—	ns
Input Data Hold Time	$t_H$	10	—	—	ns
Output Data Hold Time	$t_H$	10	25	—	ns
Address Hold Time (Address, R/W, VMA)	$t_{AH}$	50	75	—	ns
Enable High Time for DBE Input	$t_{EH}$	450	—	—	ns
Data Delay Time (Write)	$t_{DDW}$	—	165	225	ns
Processor Controls*					
Processor Control Setup Time	$t_{PCS}$	200	—	—	ns
Processor Control Rise and Fall Time	$t_{PCr}, t_{PCf}$	—	—	100	ns
Bus Available Delay	$t_{BA}$	—	—	300	ns
Three State Enable	$t_{TSE}$	—	—	40	ns
Three State Delay	$t_{TSD}$	—	—	700	ns
Data Bus Enable Down Time During $\phi 1$ Up Time (Figure 3)	$t_{DBE}$	150	—	—	ns
Data Bus Enable Delay (Figure 3)	$t_{DBED}$	300	—	—	ns
Data Bus Enable Rise and Fall Times (Figure 3)	$t_{DBEr}, t_{DBEf}$	—	—	25	ns

\*Additional information is given in Figures 12 through 16 of the Family Characteristics — see pages 17 through 20.

**FIGURE 2 — READ DATA FROM MEMORY OR PERIPHERALS**

**MOTOROLA Semiconductor Products Inc.**

## MC6800

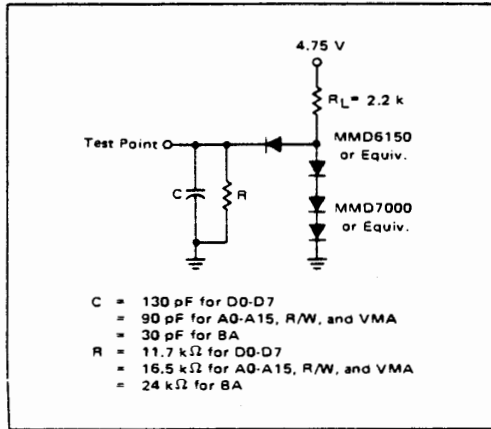
FIGURE 3 — WRITE IN MEMORY OR PERIPHERALS

FIGURE 4 — TYPICAL DATA BUS OUTPUT DELAY  
versus CAPACITIVE LOADINGFIGURE 5 — TYPICAL READ/WRITE, VMA, AND  
ADDRESS OUTPUT DELAY versus CAPACITIVE LOADING

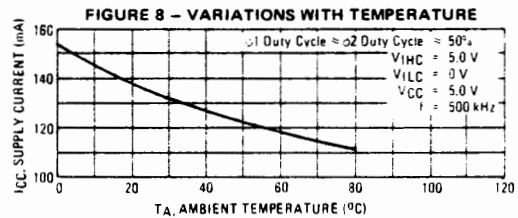
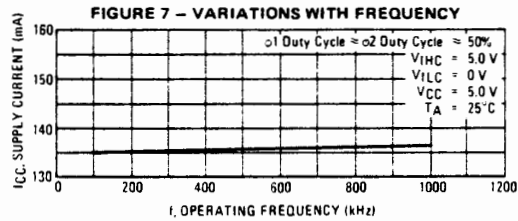
MOTOROLA Semiconductor Products Inc.

MC6800

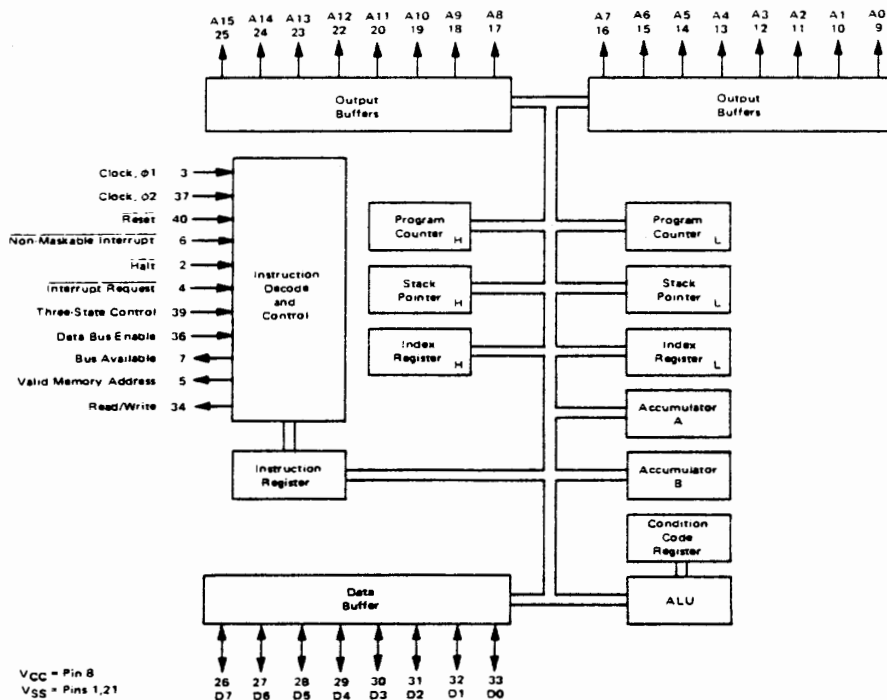
FIGURE 6 - BUS TIMING TEST LOAD



TYPICAL POWER SUPPLY CURRENT



EXPANDED BLOCK DIAGRAM



**MOTOROLA Semiconductor Products Inc.**

**MC6800**
**MPU SIGNAL DESCRIPTION**

Proper operation of the MPU requires that certain control and timing signals be provided to accomplish specific functions and that other signal lines be monitored to determine the state of the processor.

**Clocks Phase One and Phase Two ( $\phi 1, \phi 2$ )** — Two pins are used for a two-phase non-overlapping clock that runs at the VCC voltage level.

**Address Bus (A0-A15)** — Sixteen pins are used for the address bus. The outputs are three-state bus drivers capable of driving one standard TTL load and 130 pF. When the output is turned off, it is essentially an open circuit. This permits the MPU to be used in DMA applications.

**Data Bus (D0-D7)** — Eight pins are used for the data bus. It is bi-directional, transferring data to and from the memory and peripheral devices. It also has three-state output buffers capable of driving one standard TTL load and 130 pF.

**Halt** — When this input is in the low state, all activity in the machine will be halted. This input is level sensitive. In the halt mode, the machine will stop at the end of an instruction. Bus Available will be at a one level, Valid Memory Address will be at a zero, and all other three-state lines will be in the three-state mode.

Transition of the Halt line must not occur during the last 250 ns of phase one. To insure single instruction operation, the Halt line must go high for one Clock cycle.

**Three-State Control (TSC)** — This input causes all of the address lines and the Read/Write line to go into the off or high impedance state. This state will occur 700 ns after TSC = 2.0 V. The Valid Memory Address and Bus Available signals will be forced low. The data bus is not affected by TSC and has its own enable (Data Bus Enable). In DMA applications, the Three-State Control line should be brought high on the leading edge of the Phase One Clock. The  $\phi 1$  clock must be held in the high state and the  $\phi 2$  in the low state for this function to operate properly. The address bus will then be available for other devices to directly address memory. Since the MPU is a dynamic device, it can be held in this state for only 4.5  $\mu$ s or destruction of data will occur in the MPU.

**Read/Write (R/W)** — This TTL compatible output signals the peripherals and memory devices whether the MPU is in a Read (high) or Write (low) state. The normal standby state of this signal is Read (high). Three-State Control going high will turn Read/Write to the off (high impedance) state. Also, when the processor is halted, it will be in the off state. This output is capable of driving one standard TTL load and 90 pF.

**Valid Memory Address (VMA)** — This output indicates to peripheral devices that there is a valid address on the address bus. In normal operation, this signal should be utilized for enabling peripheral interfaces such as the PIA and ACIA. This signal is not three-state. One standard TTL load and 90 pF may be directly driven by this active high signal.

**Data Bus Enable (DBE)** — This input is the three-state control signal for the MPU data bus and will enable the bus drivers when in the high state. This input is TTL compatible; however in normal operation, it would be driven by the phase two clock. During an MPU read cycle, the data bus drivers will be disabled internally. When it is desired that another device control the data bus such as in Direct Memory Access (DMA) applications, DBE should be held low.

**Bus Available (BA)** — The Bus Available signal will normally be in the low state; when activated, it will go to the high state indicating that the microprocessor has stopped and that the address bus is available. This will occur if the Halt line is in the low state or the processor is in the WAIT state as a result of the execution of a WAIT instruction. At such time, all three-state output drivers will go to their off state and other outputs to their normally inactive level. The processor is removed from the WAIT state by the occurrence of a maskable (mask bit I = 0) or nonmaskable interrupt. This output is capable of driving one standard TTL load and 30 pF.

**Interrupt Request (IRQ)** — This level sensitive input requests that an interrupt sequence be generated within the machine. The processor will wait until it completes the current instruction that is being executed before it recognizes the request. At that time, if the interrupt mask bit in the Condition Code Register is not set, the machine will begin an interrupt sequence. The Index Register, Program Counter, Accumulators, and Condition Code Register are stored away on the stack. Next the MPU will respond to the interrupt request by setting the interrupt mask bit high so that no further interrupts may occur. At the end of the cycle, a 16-bit address will be loaded that points to a vectoring address which is located in memory locations FFF8 and FFF9. An address loaded at these locations causes the MPU to branch to an interrupt routine in memory.

The Halt line must be in the high state for interrupts to be serviced. Interrupts will be latched internally while Halt is low.

The IRQ has a high impedance pullup device internal to the chip; however a 3 k $\Omega$  external resistor to VCC should be used for wire-OR and optimum control of interrupts.

**Reset** — This input is used to reset and start the MPU from a power down condition, resulting from a power failure or an initial start-up of the processor. If a high level is detected on the input, this will signal the MPU to begin the restart sequence. This will start execution of a routine to initialize the processor from its reset condition. All the higher order address lines will be forced high. For the restart, the last two (FFFE, FFFF) locations in memory will be used to load the program that is addressed by the program counter. During the restart routine, the interrupt mask bit is set and must be reset before the MPU can be interrupted by IRQ.


**MOTOROLA Semiconductor Products Inc.**



**MC6800**

Figure 9 shows the initialization of the microprocessor after restart. Reset must be held low for at least eight clock periods after  $V_{CC}$  reaches 4.75 volts. If  $\overline{\text{Reset}}$  goes high prior to the leading edge of  $\phi_2$ , on the next  $\phi_1$  the first restart memory vector address (FFFE) will appear on the address lines. This location should contain the higher order eight bits to be stored into the program counter. Following, the next address FFFF should contain the lower order eight bits to be stored into the program counter.

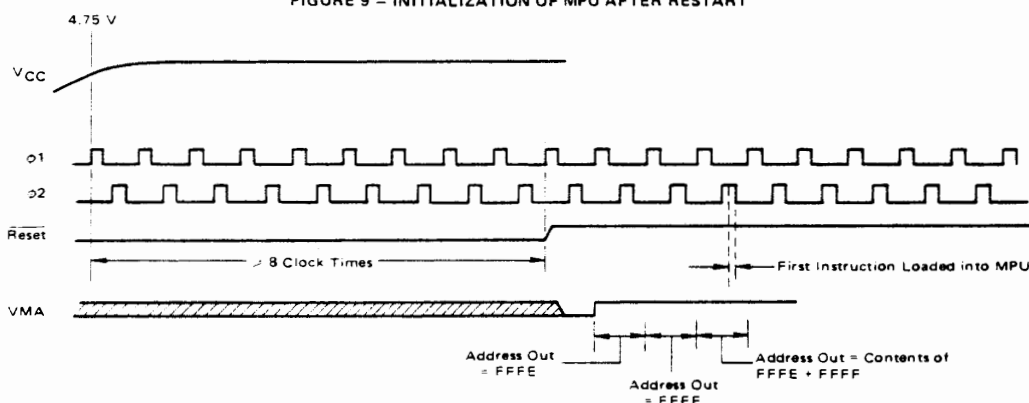
**Non-Maskable Interrupt (NMI)** — A low-going edge on this input requests that a non-mask-interrupt sequence be generated within the processor. As with the Interrupt Request signal, the processor will complete the current instruction that is being executed before it recognizes the NMI signal. The interrupt mask bit in the Condition Code Register has no effect on NMI.

The Index Register, Program Counter, Accumulators, and Condition Code Register are stored away on the stack. At the end of the cycle, a 16-bit address will be loaded that points to a vectored address which is located in memory locations FFFC and FFFD. An address loaded at these locations causes the MPU to branch to a non-maskable interrupt routine in memory.

NMI has a high impedance pullup resistor internal to the chip; however a  $3\text{ k}\Omega$  external resistor to  $V_{CC}$  should be used for wire-OR and optimum control of interrupts.

Inputs  $\overline{\text{IRQ}}$  and  $\overline{\text{NMI}}$  are hardware interrupt lines that are sampled during  $\phi_2$  and will start the interrupt routine on the  $\phi_1$  following the completion of an instruction.

Figure 10 is a flow chart describing the major decision paths and interrupt vectors of the microprocessor. Table 1 gives the memory map for interrupt vectors.

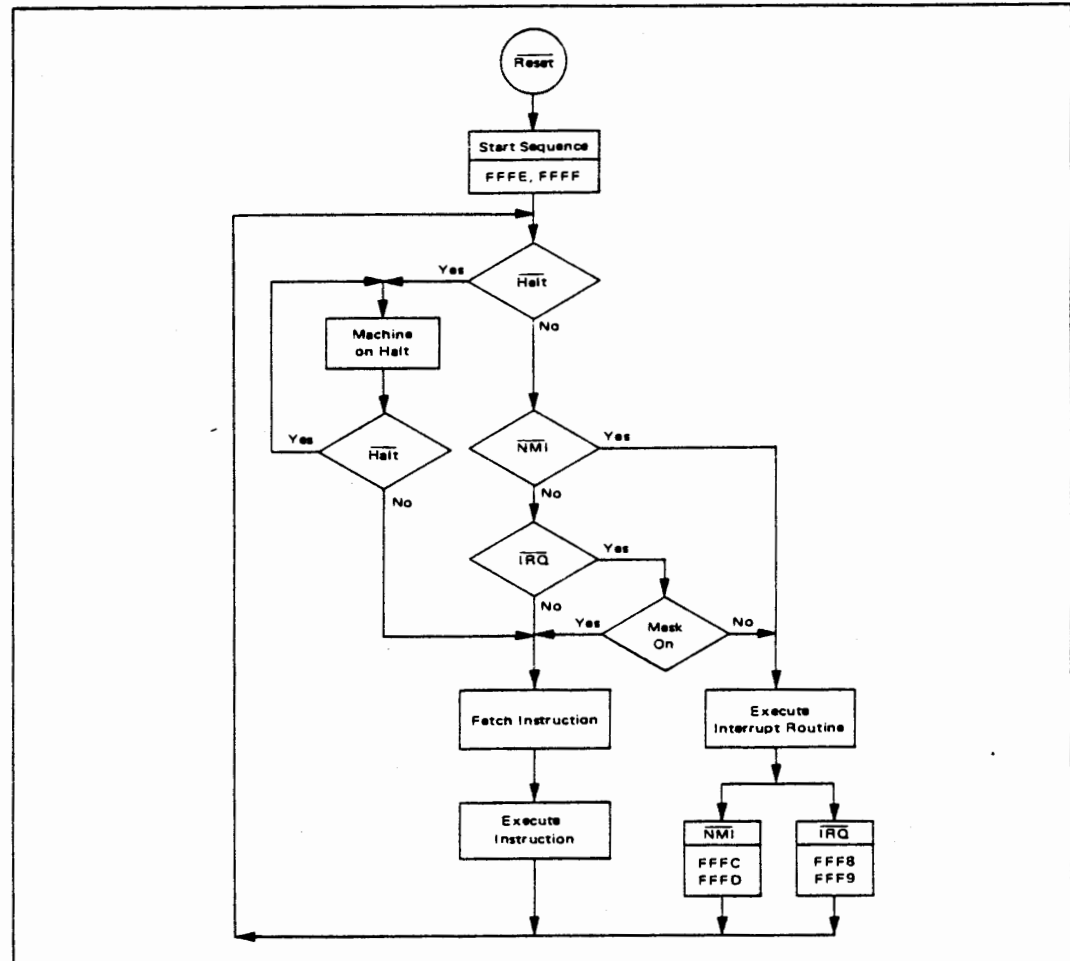
**FIGURE 9 — INITIALIZATION OF MPU AFTER RESTART**

**TABLE 1 — MEMORY MAP FOR INTERRUPT VECTORS**

Vector		Description
MS	LS	
FFFE	FFFF	Restart
FFFC	FFFD	Non-maskable Interrupt
FFFA	FFFB	Software Interrupt
FFF8	FFF9	Interrupt Request



MC6800

FIGURE 10 – MPU FLOW CHART



### MPU REGISTERS

The MPU has three 16-bit registers and three 8-bit registers available for use by the programmer (Figure 11).

**Program Counter** — The program counter is a two byte (16-bits) register that points to the current program address.

**Stack Pointer** — The stack pointer is a two byte register that contains the address of the next available location in an external push-down/pop-up stack. This stack is normally a random access Read/Write memory that may

have any location (address) that is convenient. In those applications that require storage of information in the stack when power is lost, the stack must be non-volatile.

**Index Register** — The index register is a two byte register that is used to store data or a sixteen bit memory address for the Indexed mode of memory addressing.

**Accumulators** — The MPU contains two 8-bit accumulators that are used to hold operands and results from an arithmetic logic unit (ALU).



MOTOROLA Semiconductor Products Inc.

MC6800

FIGURE 11 – PROGRAMMING MODEL OF THE MICROPROCESSING UNIT

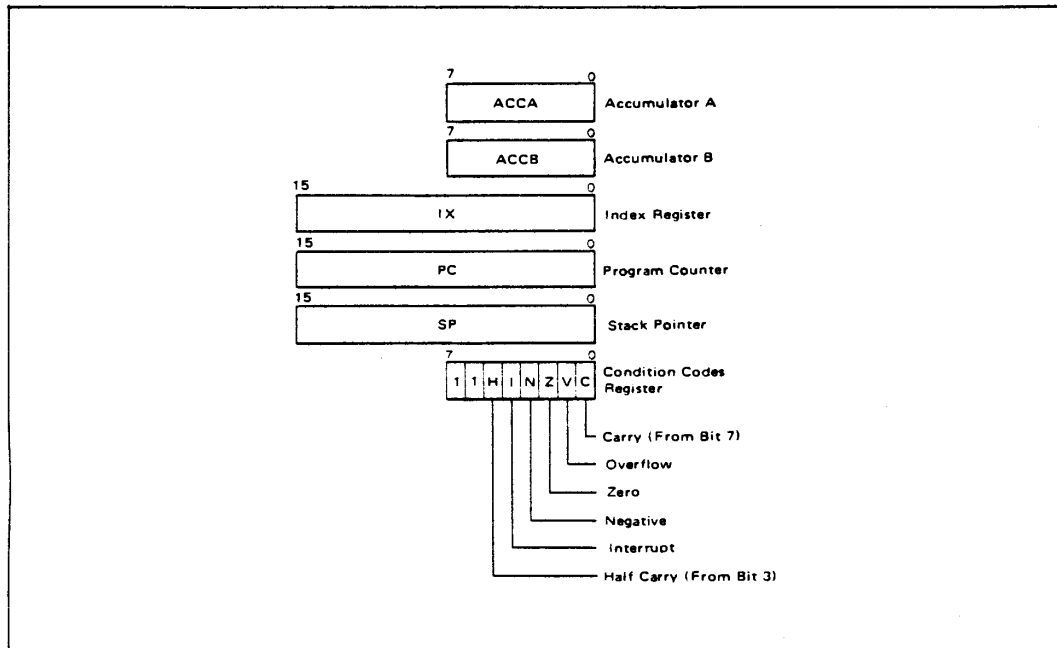
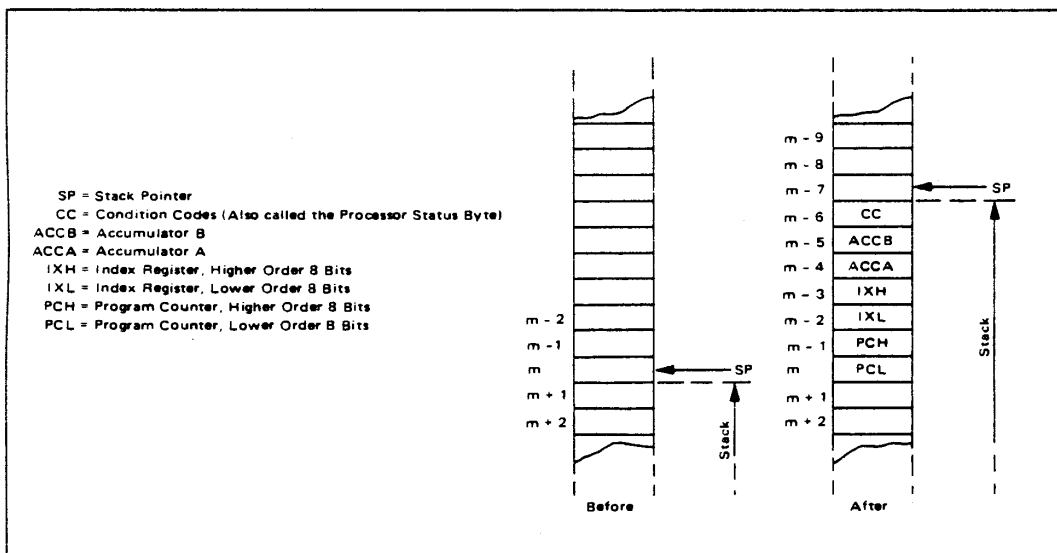


FIGURE 12 – SAVING THE STATUS OF THE MICROPROCESSOR IN THE STACK



**MOTOROLA Semiconductor Products Inc.**

**MC6800**

**Condition Code Register** — The condition code register indicates the results of an Arithmetic Logic Unit operation: Negative (N), Zero (Z), Overflow (V), Carry from bit 7 (C), and half carry from bit 3 (H). These bits of the Condition Code Register are used as testable conditions for the conditional branch instructions. Bit 4 is the interrupt mask bit (I). The unused bits of the Condition Code Register (b6 and b7) are ones.

Figure 12 shows the order of saving the microprocessor status within the stack.

**MPU INSTRUCTION SET**

The MC6800 has a set of 72 different instructions. Included are binary and decimal arithmetic, logical, shift, rotate, load, store, conditional or unconditional branch, interrupt and stack manipulation instructions (Tables 2 thru 6).

**MPU ADDRESSING MODES**

The MC6800 eight-bit microprocessing unit has seven address modes that can be used by a programmer, with the addressing mode a function of both the type of instruction and the coding within the instruction. A summary of the addressing modes for a particular instruction can be found in Table 7 along with the associated instruction execution time that is given in machine cycles. With a clock frequency of 1 MHz, these times would be microseconds.

**Accumulator (ACCX) Addressing** — In accumulator only addressing, either accumulator A or accumulator B is specified. These are one-byte instructions.

**Immediate Addressing** — In immediate addressing, the operand is contained in the second byte of the instruction except LDS and LDX which have the operand in the second and third bytes of the instruction. The MPU addresses

this location when it fetches the immediate instruction for execution. These are two or three-byte instructions.

**Direct Addressing** — In direct addressing, the address of the operand is contained in the second byte of the instruction. Direct addressing allows the user to directly address the lowest 256 bytes in the machine i.e., locations zero through 255. Enhanced execution times are achieved by storing data in these locations. In most configurations, it should be a random access memory. These are two-byte instructions.

**Extended Addressing** — In extended addressing, the address contained in the second byte of the instruction is used as the higher eight-bits of the address of the operand. The third byte of the instruction is used as the lower eight-bits of the address for the operand. This is an absolute address in memory. These are three-byte instructions.

**Indexed Addressing** — In indexed addressing, the address contained in the second byte of the instruction is added to the index register's lowest eight bits in the MPU. The carry is then added to the higher order eight bits of the index register. This result is then used to address memory. The modified address is held in a temporary address register so there is no change to the index register. These are two-byte instructions.

**Implied Addressing** — In the implied addressing mode the instruction gives the address (i.e., stack pointer, index register, etc.). These are one-byte instructions.

**Relative Addressing** — In relative addressing, the address contained in the second byte of the instruction is added to the program counter's lowest eight bits plus two. The carry or borrow is then added to the high eight bits. This allows the user to address data within a range of -125 to +129 bytes of the present instruction. These are two-byte instructions.

TABLE 2 — MICROPROCESSOR INSTRUCTION SET — ALPHABETIC SEQUENCE

ABA	Add Accumulators	CLR	Clear	PUL	Pull Data
ADC	Add with Carry	CLV	Clear Overflow	ROL	Rotate Left
ADD	Add	CMP	Compare	ROR	Rotate Right
AND	Logical And	COM	Complement	RTI	Return from Interrupt
ASL	Arithmetic Shift Left	CPX	Compare Index Register	RTS	Return from Subroutine
ASR	Arithmetic Shift Right	DAA	Decimal Adjust	SBA	Subtract Accumulators
BCC	Branch if Carry Clear	DEC	Decrement	SBC	Subtract with Carry
BCS	Branch if Carry Set	DES	Decrement Stack Pointer	SEC	Set Carry
BEQ	Branch if Equal to Zero	DEX	Decrement Index Register	SEI	Set Interrupt Mask
BGE	Branch if Greater or Equal Zero	EOR	Exclusive OR	SEV	Set Overflow
BGT	Branch if Greater than Zero	INC	Increment	STA	Store Accumulator
BHI	Branch if Higher	INS	Increment Stack Pointer	STS	Store Stack Register
BIT	Bit Test	INX	Increment Index Register	STX	Store Index Register
BLE	Branch if Less or Equal	JMP	Jump	SUB	Subtract
BLS	Branch if Lower or Same	JSR	Jump to Subroutine	SWI	Software Interrupt
BLT	Branch if Less than Zero	LDA	Load Accumulator	TAB	Transfer Accumulators
BMI	Branch if Minus	LDS	Load Stack Pointer	TAP	Transfer Accumulators to Condition Code Reg.
BNE	Branch if Not Equal to Zero	LDX	Load Index Register	TBA	Transfer Accumulators
BPL	Branch if Plus	LSR	Logical Shift Right	TPA	Transfer Condition Code Reg. to Accumulator
BRA	Branch Always	NEG	Negate	TST	Test
BSR	Branch to Subroutine	NOP	No Operation	TSX	Transfer Stack Pointer to Index Register
BVC	Branch if Overflow Clear	ORA	Inclusive OR Accumulator	TXS	Transfer Index Register to Stack Pointer
BVS	Branch if Overflow Set	PSH	Push Data	WAI	Wait for Interrupt
CBA	Compare Accumulators				
CLC	Clear Carry				
CLI	Clear Interrupt Mask				



**MOTOROLA Semiconductor Products Inc.**

**MC6800**

**TABLE 3 - ACCUMULATOR AND MEMORY INSTRUCTIONS**

OPERATIONS	MNEMONIC	ADDRESSING MODES					BOOLEAN/ARITHMETIC OPERATION (All register labels refer to contents)	COND. CODE REG.					
		IMMED	DIRECT	INDEX	EXTND	IMPLIED		S	O	N	Z	V	C
Add	ADD	38 2 2	38 3 2	A8 5 2	B8 4 3		A ← M + A	•	•	•	•	•	•
Add	ADD	CB 2 2	CB 3 2	E8 5 2	F8 4 3		B ← M + B	•	•	•	•	•	•
Add Accum	ABA					18 2 1	A ← B + A	•	•	•	•	•	•
Add with Carry	ADCA	39 2 2	39 3 2	A9 5 2	B9 4 3		A ← M + C + A	•	•	•	•	•	•
	ADCB	CB 2 2	CB 3 2	E9 5 2	F9 4 3		B ← M + C + B	•	•	•	•	•	•
And	ANDA	34 2 2	34 3 2	A4 5 2	B4 4 3		A ← M & A	•	•	•	•	•	•
	ANDB	CA 2 2	CA 3 2	E4 5 2	F4 4 3		B ← M & B	•	•	•	•	•	•
Bit Test	BITA	35 2 2	35 3 2	A5 5 2	B5 4 3		A ← M	•	•	•	•	•	•
	BITB	CA 2 2	CA 3 2	E5 5 2	F5 4 3		B ← M	•	•	•	•	•	•
Clear	CLR			6F 7 2	7F 6 3		00 ← M	•	•	•	•	•	•
	CLRA					2F 2 1	00 ← A	•	•	•	•	•	•
	CLRB					3F 2 1	00 ← B	•	•	•	•	•	•
Compare	CPA	31 2 2	31 3 2	A1 5 2	B1 4 3		A ← M	•	•	•	•	•	•
	CPMB	CB 2 2	CB 3 2	E1 5 2	F1 4 3		B ← M	•	•	•	•	•	•
Compare Accum	CBA					11 2 1	A ← B	•	•	•	•	•	•
Complement B	CMB			63 7 2	73 6 3		M ← M	•	•	•	•	•	•
	COMB					43 2 1	A ← A	•	•	•	•	•	•
	COMB					53 2 1	B ← B	•	•	•	•	•	•
Complement 2's	NEG			60 7 2	70 6 3		00 ← M	•	•	•	•	•	•
Negate	NEGA					40 2 1	00 ← A	•	•	•	•	•	•
	NEGB					50 2 1	00 ← B	•	•	•	•	•	•
Decimal Adjust A	DAA					19 2 1	Converts binary Add of BCD Characters into BCD Format	•	•	•	•	•	•
Decrement	DEC			5A 7 2	7A 6 3		M ← M	•	•	•	•	•	•
	DECA					1A 2 1	A ← A	•	•	•	•	•	•
	DECB					2A 2 1	B ← B	•	•	•	•	•	•
Exchange R/R	EXR	38 2 2	38 3 2	A8 5 2	B8 4 3		A ↔ B	•	•	•	•	•	•
	EXRB	CA 2 2	CA 3 2	E8 5 2	F8 4 3		B ↔ A	•	•	•	•	•	•
Increment	INC			6C 7 2	7C 6 3		M ← M	•	•	•	•	•	•
	INCA					1C 2 1	A ← A	•	•	•	•	•	•
	INCB					2C 2 1	B ← B	•	•	•	•	•	•
Load Accum	LDA	36 2 2	36 3 2	A6 5 2	B6 4 3		M ← A	•	•	•	•	•	•
	LDAB	CB 2 2	CB 3 2	E6 5 2	F6 4 3		M ← B	•	•	•	•	•	•
Load Register	LRAA	3A 2 2	3A 3 2	AA 5 2	BA 4 3		A ← M + A	•	•	•	•	•	•
	LRAB	CA 2 2	CA 3 2	EA 5 2	FA 4 3		B ← M + B	•	•	•	•	•	•
Push Data	PSHA					76 4 1	A ← Msp - 1 - SP	•	•	•	•	•	•
	PSHB					77 4 1	B ← Msp - 1 - SP	•	•	•	•	•	•
Pop Data	PULA					32 4 1	SP ← 1 - SP Msp - A	•	•	•	•	•	•
	PULB					33 4 1	SP ← 1 - SP Msp - B	•	•	•	•	•	•
Rotate Left	ROL			59 7 2	79 6 3		M	•	•	•	•	•	•
	ROLA					29 2 1	A ← Msp - 1 - SP	•	•	•	•	•	•
	ROLB					39 2 1	B ← Msp - 1 - SP	•	•	•	•	•	•
Rotate Right	ROR			66 7 2	76 6 3		M	•	•	•	•	•	•
	RORA					16 2 1	A ← Msp - 1 - SP	•	•	•	•	•	•
	RORB					26 2 1	B ← Msp - 1 - SP	•	•	•	•	•	•
Shift Left Arithmetic	ASL			58 7 2	78 6 3		M	•	•	•	•	•	•
	ASLA					18 2 1	A ← Msp - 1 - SP	•	•	•	•	•	•
	ASLB					28 2 1	B ← Msp - 1 - SP	•	•	•	•	•	•
Shift Right Arithmetic	ASR			57 7 2	77 6 3		M	•	•	•	•	•	•
	ASRA					17 2 1	A ← Msp - 1 - SP	•	•	•	•	•	•
	ASRB					27 2 1	B ← Msp - 1 - SP	•	•	•	•	•	•
Shift Right Logical	LSR			64 7 2	74 6 3		M	•	•	•	•	•	•
	LSRA					14 2 1	A ← Msp - 1 - SP	•	•	•	•	•	•
	LSRB					24 2 1	B ← Msp - 1 - SP	•	•	•	•	•	•
Store Accum	STA		37 4 2	A7 5 2	B7 4 3		A ← M	•	•	•	•	•	•
	STAB		CB 4 2	E7 5 2	F7 4 3		B ← M	•	•	•	•	•	•
Subtract	SUBA	40 2 2	40 3 2	A0 5 2	B0 4 3		A ← M - A	•	•	•	•	•	•
	SUBB	CA 2 2	CA 3 2	E0 5 2	F0 4 3		B ← M - B	•	•	•	•	•	•
Subtract Accum	SBA					10 2 1	A ← B - A	•	•	•	•	•	•
Subtr with Carry	SBCA	32 2 2	32 3 2	A2 5 2	B2 4 3		A ← M - C - A	•	•	•	•	•	•
	SBCB	CA 2 2	CA 3 2	E2 5 2	F2 4 3		B ← M - C - B	•	•	•	•	•	•
Transfer Accum	TBA					16 2 1	A ← B	•	•	•	•	•	•
	TAB					26 2 1	B ← A	•	•	•	•	•	•
Test Zero or Minus	TST			60 7 2	70 6 3		M - 00	•	•	•	•	•	•
	TSTA					40 2 1	A - 00	•	•	•	•	•	•
	TSTB					50 2 1	B - 00	•	•	•	•	•	•

**LEGEND:**

- OP - Operation Code (Hexadecimal)
- Number of MPU Cycles
- Number of Program Bytes
- Arithmetic Plus
- Arithmetic Minus
- Boolean AND
- Msp - Contents of memory location pointed to by Stack Pointer
- Note - Accumulator addressing mode instructions are included in the column for IMPLIED addressing
- Boolean Inclusive OR
- Boolean Exclusive OR
- Complement of M
- Transfer into M
- 0 Bit Zero
- 00 Byte Zero

**CONDITION CODE SYMBOLS:**

- H - Half Carry from bit 3
- I - Interrupt mask
- N - Negative from bit 7
- Z - Zero flag
- V - Overflow 2's Complement
- C - Carry from bit 7
- R - Register Always
- S - Set Always
- Test and set if true, cleared otherwise
- Not Affected



**MOTOROLA Semiconductor Products Inc.**

## MC6800

TABLE 4 — INDEX REGISTER AND STACK MANIPULATION INSTRUCTIONS

																	COND. CODE REG.						
		IMMED			DIRECT			INDEX			EXTND			IMPLIED			BOOLEAN/ARITHMETIC OPERATION						
POINTER OPERATIONS	MNEMONIC	OP	~	=	OP	~	=	OP	~	=	OP	~	=	OP	~	=	H	I	N	Z	V	C	
Compare Index Reg	CPX	8C	3	3	9C	4	2	AC	6	2	BC	5	3				$X_H - M, X_L - (M + 1)$	•	•	⑦	•	•	•
Decrement Index Reg	DEX													09	4	1	$X - 1 \rightarrow X$	•	•	•	•	•	•
Decrement Stack Ptr	DES													34	4	1	$SP - 1 \rightarrow SP$	•	•	•	•	•	•
Increment Index Reg	INX													08	4	1	$X + 1 \rightarrow X$	•	•	•	•	•	•
Increment Stack Ptr	INS													31	4	1	$SP + 1 \rightarrow SP$	•	•	•	•	•	•
Load Index Reg	LDX	CE	3	3	DE	4	2	EE	6	2	FE	5	3				$M \rightarrow X_H, (M + 1) \rightarrow X_L$	•	•	③	•	•	R
Load Stack Ptr	LOS	8E	3	3	9E	4	2	AE	6	2	BE	5	3				$M \rightarrow SP_H, (M + 1) \rightarrow SP_L$	•	•	④	•	•	R
Store Index Reg	STX				DF	5	2	EF	7	2	FF	6	3				$X_H \rightarrow M, X_L \rightarrow (M + 1)$	•	•	•	•	•	•
Store Stack Ptr	STS				9F	5	2	AF	7	2	8F	6	3				$SP_H \rightarrow M, SP_L \rightarrow (M + 1)$	•	•	⑨	•	•	R
Index Reg → Stack Ptr	TXS													35	4	1	$X - 1 \rightarrow SP$	•	•	•	•	•	•
Stack Ptr → Index Reg	TSX													30	4	1	$SP + 1 \rightarrow X$	•	•	•	•	•	•

TABLE 5 — JUMP AND BRANCH INSTRUCTIONS

OPERATIONS	MNEMONIC													COND. CODE REG.					
		RELATIVE			INDEX			EXTND			IMPLIED			BRANCH TEST					
		OP	~	=	OP	~	=	OP	~	=	OP	~	=	H	I	N	Z	V	C
Branch Always	BRA	20	4	2										None	•	•	•	•	•
Branch If Carry Clear	BCC	24	4	2										C = 0	•	•	•	•	•
Branch If Carry Set	BCS	25	4	2										C = 1	•	•	•	•	•
Branch If = Zero	BEQ	27	4	2										Z = 1	•	•	•	•	•
Branch If > Zero	BGE	2C	4	2										N ⊕ V = 0	•	•	•	•	•
Branch If > Zero	BGT	2E	4	2										Z + (N ⊕ V) = 0	•	•	•	•	•
Branch If Higher	BHI	22	4	2										C + Z = 0	•	•	•	•	•
Branch If < Zero	BLE	2F	4	2										Z + (N ⊕ V) = 1	•	•	•	•	•
Branch If Lower Or Same	BLS	23	4	2										C + Z = 1	•	•	•	•	•
Branch If < Zero	BLT	2D	4	2										N ⊕ V = 1	•	•	•	•	•
Branch If Minus	BMI	28	4	2										N = 1	•	•	•	•	•
Branch If Not Equal Zero	BNE	26	4	2										Z = 0	•	•	•	•	•
Branch If Overflow Clear	BVC	28	4	2										V = 0	•	•	•	•	•
Branch If Overflow Set	BVS	29	4	2										V = 1	•	•	•	•	•
Branch If Plus	BPL	2A	4	2										N = 0	•	•	•	•	•
Branch To Subroutine	BSR	8D	8	2											•	•	•	•	•
Jump	JMP				6E	4	2	7E	3	3				See Special Operations	•	•	•	•	•
Jump To Subroutine	JSR				AD	8	2	8D	9	3					•	•	•	•	•
No Operation	NOP										01	2	1	Advances Prog. Cntr. Only	•	•	•	•	•
Return From Interrupt	RTI										38	10	1		•	•	•	•	•
Return From Subroutine	RTS										39	5	1		•	•	•	•	•
Software Interrupt	SWI										3F	12	1	See Special Operations	•	•	•	•	•
Wait for Interrupt*	WAI										3E	9	1		•	•	•	•	•

\*WAI puts Address Bus, R/W, and Data Bus in the three-state mode while VMA is held low.

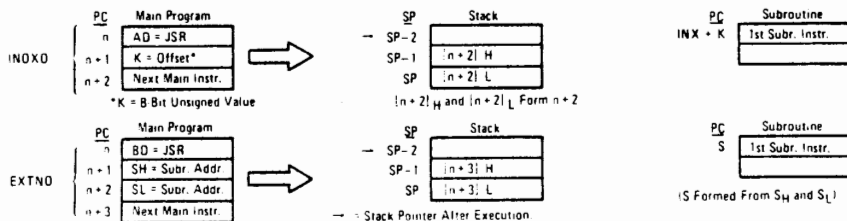


MOTOROLA Semiconductor Products Inc.

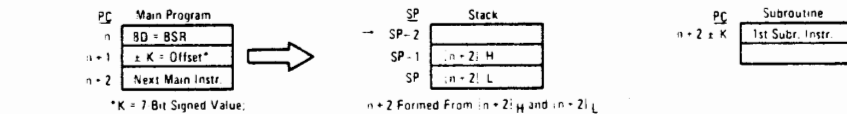
## MC6800

## SPECIAL OPERATIONS

## JSR, JUMP TO SUBROUTINE:



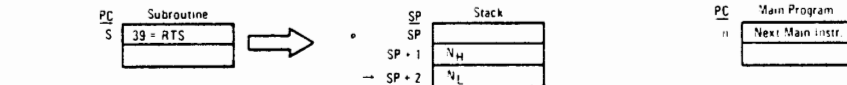
## BSR, BRANCH TO SUBROUTINE:



## JMP, JUMP:



## RTS, RETURN FROM SUBROUTINE:



## RTI, RETURN FROM INTERRUPT:

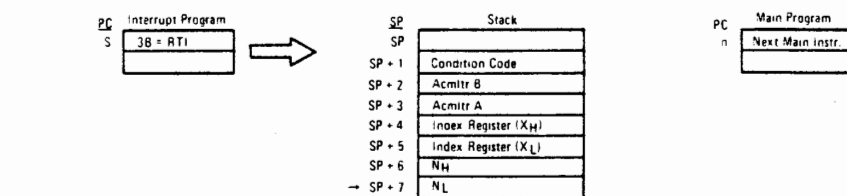


TABLE 6 - CONDITION CODE REGISTER MANIPULATION INSTRUCTIONS

OPERATIONS	MNEMONIC	IMPLIED	OP	~	=	BOOLEAN OPERATION	COND. CODE REG.						
							5	4	3	2	1	0	
							H	I	N	Z	V	C	
Clear Carry	CLC	0C	2	1		0 ← C	•	•	•	•	•	•	R
Clear Interrupt Mask	CLI	0E	2	1		0 ← I	•	•	•	•	•	•	•
Clear Overflow	CLV	0A	2	1		0 ← V	•	•	•	•	•	•	R
Set Carry	SEC	0D	2	1		1 ← C	•	•	•	•	•	•	S
Set Interrupt Mask	SEI	0F	2	1		1 ← I	•	S	•	•	•	•	•
Set Overflow	SEV	0B	2	1		1 ← V	•	•	•	•	•	•	S
Accmtr A → CCR	TAP	06	2	1		A → CCR	12						
CCR → Accmtr A	TPA	07	2	1		CCR → A	•	•	•	•	•	•	•

CONDITION CODE REGISTER NOTES: (Bit set if test is true and cleared otherwise)

- (Bit V) Test: Result = 10000000?
- (Bit C) Test: Result = 00000000?
- (Bit C) Test: Decimal value of most significant BCD Character greater than nine? (Not cleared if previously set.)
- (Bit V) Test: Operand = 10000000 prior to execution?
- (Bit V) Test: Operand = 01111111 prior to execution?
- (Bit V) Test: Set equal to result of NOC after shift has occurred.
- (Bit N) Test: Sign bit of most significant (MS) byte = 1?
- (Bit V) Test: 2's complement overflow from subtraction of MS bytes?
- (Bit N) Test: Result less than zero? (Bit 15 = 1)
- (All) Load Condition Code Register from Stack. (See Special Operations)
- (Bit I) Set when interrupt occurs. If previously set, a Non-Maskable Interrupt is required to exit the wait state.
- (All) Set according to the contents of Accumulator A.



MOTOROLA Semiconductor Products Inc.

## MC6800

TABLE 7 - INSTRUCTION ADDRESSING MODES AND ASSOCIATED EXECUTION TIMES  
(Times in Machine Cycles)

	(Dual Operand)							(Dual Operand)					
	ACCX	Immediate	Direct	Extended	Indexed	Implied		ACCX	Immediate	Direct	Extended	Indexed	Implied
ABA							INC						
ADC	x		2	3	4	5	INS	2			6	7	
ADD	x		2	3	4	5	INX						4
AND	x		2	3	4	5	JMP				3	4	
ASL	2			6	7		JSR				9	8	
ASR	2			6	7		LDA	x	2	3	4	5	
BCC						4	LDS		3	4	5	6	
BES						4	LDX		3	4	5	6	
BEA						4	LSR	2			6	7	
BGE						4	NEG	2			6	7	
BGT						4	NOP						2
BHI						4	ORA	x	2	3	4	5	
BIT	x	2	3	4	5		PSH						4
BLE						4	PUL						4
BLS						4	RCL	2			6	7	
BLT						4	ROR	2			6	7	
BMI						4	RTI						10
BNE						4	RTS						5
BPL						4	SBA						2
BRA						4	SBC	x	2	3	4	5	
BSR						8	SEC						2
BVC						4	SEI						2
BVS						4	SEV						2
CBA					2		STA	x		4	5	6	
CLC					2		STS			5	6	7	
CLI					2		STX			5	6	7	
CLR	2			6	7		SUB	x	2	3	4	5	
CLV					2		SWI						12
CMP	x	2	3	4	5		TAB						2
COM		2		6	7		TAP						2
CPX		3	4	5	6		TBA						2
DAA					2		TPA						2
DEC	2			6	7		TST	2		6	7		
DES					4		TSX						4
DEX					4		TSX						4
EOR	x	2	3	4	5		WAI						9

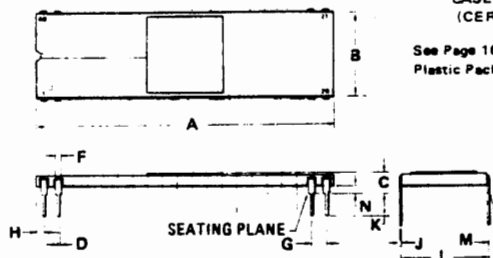
NOTE Interrupt time is 12 cycles from the end of the instruction being executed, except following a WAI instruction. Then it is 4 cycles.

## PIN ASSIGNMENT

1	$\overline{V_{SS}}$	Reset	40
2	Halt	TSC	39
3	$\phi 1$	N.C.	38
4	IRQ	$\phi 2$	37
5	VMA	DBE	36
6	NMI	N.C.	35
7	BA	R/W	34
8	VCC	D0	33
9	A0	D1	32
10	A1	D2	31
11	A2	D3	30
12	A3	D4	29
13	A4	D5	28
14	A5	D6	27
15	A6	D7	26
16	A7	A15	25
17	A8	A14	24
18	A9	A13	23
19	A10	A12	22
20	A11	VSS	21

PACKAGE DIMENSIONS  
CASE 715-02  
(CERAMIC)

See Page 165 for  
Plastic Package dimensions.



DIM	MILLIMETERS		INCHES	
	MIN	MAX	MIN	MAX
A	50.29	51.31	1.980	2.020
B	14.86	15.62	0.585	0.615
C	2.54	4.19	0.100	0.165
D	0.38	0.53	0.015	0.021
F	0.76	1.40	0.030	0.055
G	2.54 BSC		0.100 BSC	
H	0.76	1.78	0.030	0.070
J	0.20	0.33	0.008	0.013
K	2.54	4.19	0.100	0.165
L	14.60	15.37	0.575	0.605
M	10 <sup>0</sup>		10 <sup>0</sup>	
N	0.51	1.52	0.020	0.060

NOTE:  
1. LEADS, TRUE POSITIONED WITHIN  
0.25 mm (0.010) DIA (AT SEATING  
PLANE), AT MAX. MAT'L  
CONDITION.



MOTOROLA Semiconductor Products Inc.



**MC6800**
**SUMMARY OF CYCLE BY CYCLE OPERATION**

Table 8 provides a detailed description of the information present on the Address Bus, Data Bus, Valid Memory Address line (VMA), and the Read/Write line (R/W) during each cycle for each instruction.

This information is useful in comparing actual with expected results during debug of both software and hard-

ware as the control program is executed. The information is categorized in groups according to Addressing Mode and Number of Cycles per instruction. (In general, instructions with the same Addressing Mode and Number of Cycles execute in the same manner; exceptions are indicated in the table.)

**TABLE 8 - OPERATION SUMMARY**

Address Mode and Instructions	Cycles	Cycle =	VMA Line	Address Bus	R/W Line	Data Bus
<b>IMMEDIATE</b>						
ADC EOR ADD LDA AND ORA BIT SBC CMP SUB	2	1 2	1 1	Op Code Address Op Code Address + 1	1 1	Op Code Operand Data
CPX LDS LDX	3	1 2 3	1 1 1	Op Code Address Op Code Address + 1 Op Code Address + 2	1 1 1	Op Code Operand Data (High Order Byte) Operand Data (Low Order Byte)
<b>DIRECT</b>						
ADC EOR ADD LDA AND ORA BIT SBC CMP SUB	3	1 2 3	1 1 1	Op Code Address Op Code Address + 1 Address of Operand	1 1 1	Op Code Address of Operand Operand Data
CPX LDS LDX	4	1 2 3 4	1 1 1 1	Op Code Address Op Code Address + 1 Address of Operand Operand Address + 1	1 1 1 1	Op Code Address of Operand Operand Data (High Order Byte) Operand Data (Low Order Byte)
STA	4	1 2 3 4	1 1 0 1	Op Code Address Op Code Address + 1 Destination Address Destination Address	1 1 1 0	Op Code Destination Address Irrelevant Data (Note 1) Data from Accumulator
STS STX	5	1 2 3 4 5	1 1 0 1 1	Op Code Address Op Code Address + 1 Address of Operand Address of Operand Address of Operand + 1	1 1 1 0 0	Op Code Address of Operand Irrelevant Data (Note 1) Register Data (High Order Byte) Register Data (Low Order Byte)
<b>INDEXED</b>						
JMP	4	1 2 3 4	1 1 0 0	Op Code Address Op Code Address + 1 Index Register Index Register Plus Offset (w/o Carry)	1 1 1 1	Op Code Offset Irrelevant Data (Note 1) Irrelevant Data (Note 1)
ADC EOR ADD LDA AND ORA BIT SBC CMP SUB	5	1 2 3 4 5	1 1 0 0 1	Op Code Address Op Code Address + 1 Index Register Index Register Plus Offset (w/o Carry) Index Register Plus Offset	1 1 1 1 1	Op Code Offset Irrelevant Data (Note 1) Irrelevant Data (Note 1) Operand Data
CPX LDS LDX	6	1 2 3 4 5 6	1 1 0 0 1 1	Op Code Address Op Code Address + 1 Index Register Index Register Plus Offset (w/o Carry) Index Register Plus Offset Index Register Plus Offset + 1	1 1 1 1 1 1	Op Code Offset Irrelevant Data (Note 1) Irrelevant Data (Note 1) Operand Data (High Order Byte) Operand Data (Low Order Byte)



## MC6800

TABLE 8 — OPERATION SUMMARY (Continued)

Address Mode and Instructions	Cycles	Cycle #	VMA Line	Address Bus	R/W Line	Data Bus
INDEXED (Continued)						
STA	6	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Offset
		3	0	Index Register	1	Irrelevant Data (Note 1)
		4	0	Index Register Plus Offset (w/o Carry)	1	Irrelevant Data (Note 1)
		5	0	Index Register Plus Offset	1	Irrelevant Data (Note 1)
		6	1	Index Register Plus Offset	0	Operand Data
ASL LSR ASR NEG CLR ROL COM ROR DEC TST INC	7	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Offset
		3	0	Index Register	1	Irrelevant Data (Note 1)
		4	0	Index Register Plus Offset (w/o Carry)	1	Irrelevant Data (Note 1)
		5	1	Index Register Plus Offset	1	Current Operand Data
		6	0	Index Register Plus Offset	1	Irrelevant Data (Note 1)
		7	1/0 (Note 3)	Index Register Plus Offset	0	New Operand Data (Note 3)
STS STX	7	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Offset
		3	0	Index Register	1	Irrelevant Data (Note 1)
		4	0	Index Register Plus Offset (w/o Carry)	1	Irrelevant Data (Note 1)
		5	0	Index Register Plus Offset	1	Irrelevant Data (Note 1)
		6	1	Index Register Plus Offset	0	Operand Data (High Order Byte)
		7	1	Index Register Plus Offset + 1	0	Operand Data (Low Order Byte)
JSR	8	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Offset
		3	0	Index Register	1	Irrelevant Data (Note 1)
		4	1	Stack Pointer	0	Return Address (Low Order Byte)
		5	1	Stack Pointer - 1	0	Return Address (High Order Byte)
		6	0	Stack Pointer - 2	1	Irrelevant Data (Note 1)
		7	0	Index Register	1	Irrelevant Data (Note 1)
		8	0	Index Register Plus Offset (w/o Carry)	1	Irrelevant Data (Note 1)
EXTENDED						
JMP	3	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Jump Address (High Order Byte)
		3	1	Op Code Address + 2	1	Jump Address (Low Order Byte)
ADC EOR ADD LDA AND ORA BIT SBC CMP SUB	4	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Address of Operand (High Order Byte)
		3	1	Op Code Address + 2	1	Address of Operand (Low Order Byte)
		4	1	Address of Operand	1	Operand Data
CPX LDS LDX	5	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Address of Operand (High Order Byte)
		3	1	Op Code Address + 2	1	Address of Operand (Low Order Byte)
		4	1	Address of Operand	1	Operand Data (High Order Byte)
		5	1	Address of Operand + 1	1	Operand Data (Low Order Byte)
STA A STA B	5	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Destination Address (High Order Byte)
		3	1	Op Code Address + 2	1	Destination Address (Low Order Byte)
		4	0	Operand Destination Address	1	Irrelevant Data (Note 1)
		5	1	Operand Destination Address	0	Data from Accumulator
ASL LSR ASR NEG CLR ROL COM ROR DEC TST INC	6	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Address of Operand (High Order Byte)
		3	1	Op Code Address + 2	1	Address of Operand (Low Order Byte)
		4	1	Address of Operand	1	Current Operand Data
		5	0	Address of Operand	1	Irrelevant Data (Note 1)
		6	1/0 (Note 3)	Address of Operand	0	New Operand Data (Note 3)

**MC6800**
**TABLE 8 - OPERATION SUMMARY (Continued)**

Address Mode and Instructions	Cycles	Cycle =	VMA Line	Address Bus	R/W Line	Data Bus
EXTENDED (Continued)						
STS STX	6	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Address of Operand (High Order Byte)
		3	1	Op Code Address + 2	1	Address of Operand (Low Order Byte)
		4	0	Address of Operand	1	Irrelevant Data (Note 1)
		5	1	Address of Operand	0	Operand Data (High Order Byte)
		6	1	Address of Operand + 1	0	Operand Data (Low Order Byte)
JSR	9	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Address of Subroutine (High Order Byte)
		3	1	Op Code Address + 2	1	Address of Subroutine (Low Order Byte)
		4	1	Subroutine Starting Address	1	Op Code of Next Instruction
		5	1	Stack Pointer	0	Return Address (Low Order Byte)
		6	1	Stack Pointer - 1	0	Return Address (High Order Byte)
		7	0	Stack Pointer - 2	1	Irrelevant Data (Note 1)
		8	0	Op Code Address + 2	1	Irrelevant Data (Note 1)
		9	1	Op Code Address + 2	1	Address of Subroutine (Low Order Byte)
INHERENT						
ABA DAA SEC ASL DEC SEI ASR INC SEV CBA LSR TAB CLC NEG TAP CLI NOP TBA CLR ROL TPA CLV ROR TST COM SBA	2	1	1	Op Code Address Op Code Address + 1	1	Op Code Op Code of Next Instruction
DES DEX INS INX	4	1	1	Op Code Address Op Code Address + 1 Previous Register Contents New Register Contents	1 1 1 1	Op Code Op Code of Next Instruction Irrelevant Data (Note 1) Irrelevant Data (Note 1)
PSH	4	1	1	Op Code Address Op Code Address + 1 Stack Pointer Stack Pointer - 1	1 1 0 1	Op Code Op Code of Next Instruction Accumulator Data Accumulator Data
PUL	4	1	1	Op Code Address Op Code Address + 1 Stack Pointer Stack Pointer + 1	1 1 1 1	Op Code Op Code of Next Instruction Irrelevant Data (Note 1) Operand Data from Stack
TSX	4	1	1	Op Code Address Op Code Address + 1 Stack Pointer New Index Register	1 1 1 1	Op Code Op Code of Next Instruction Irrelevant Data (Note 1) Irrelevant Data (Note 1)
TXS	4	1	1	Op Code Address Op Code Address + 1 Index Register New Stack Pointer	1 1 1 1	Op Code Op Code of Next Instruction Irrelevant Data Irrelevant Data
RTS	5	1	1	Op Code Address Op Code Address + 1 Stack Pointer Stack Pointer + 1 Stack Pointer + 2	1 1 1 1 1	Op Code Irrelevant Data (Note 2) Irrelevant Data (Note 1) Address of Next Instruction (High Order Byte) Address of Next Instruction (Low Order Byte)



## MC6800

TABLE 8 - OPERATION SUMMARY (Continued)

Address Mode and Instructions	Cycles	Cycle #	VMA Line	Address Bus	R/W Line	Data Bus
INHERENT (Continued)						
WAI	9	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Op Code of Next Instruction
		3	1	Stack Pointer	0	Return Address (Low Order Byte)
		4	1	Stack Pointer - 1	0	Return Address (High Order Byte)
		5	1	Stack Pointer - 2	0	Index Register (Low Order Byte)
		6	1	Stack Pointer - 3	0	Index Register (High Order Byte)
		7	1	Stack Pointer - 4	0	Contents of Accumulator A
		8	1	Stack Pointer - 5	0	Contents of Accumulator B
		9	1	Stack Pointer - 6 (Note 4)	1	Contents of Cond. Code Register
RTI	10	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Irrelevant Data (Note 2)
		3	0	Stack Pointer	1	Irrelevant Data (Note 1)
		4	1	Stack Pointer + 1	1	Contents of Cond. Code Register from Stack
		5	1	Stack Pointer + 2	1	Contents of Accumulator B from Stack
		6	1	Stack Pointer + 3	1	Contents of Accumulator A from Stack
		7	1	Stack Pointer + 4	1	Index Register from Stack (High Order Byte)
		8	1	Stack Pointer + 5	1	Index Register from Stack (Low Order Byte)
		9	1	Stack Pointer + 6	1	Next Instruction Address from Stack (High Order Byte)
		10	1	Stack Pointer + 7	1	Next Instruction Address from Stack (Low Order Byte)
SWI	12	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Irrelevant Data (Note 1)
		3	1	Stack Pointer	0	Return Address (Low Order Byte)
		4	1	Stack Pointer - 1	0	Return Address (High Order Byte)
		5	1	Stack Pointer - 2	0	Index Register (Low Order Byte)
		6	1	Stack Pointer - 3	0	Index Register (High Order Byte)
		7	1	Stack Pointer - 4	0	Contents of Accumulator A
		8	1	Stack Pointer - 5	0	Contents of Accumulator B
		9	1	Stack Pointer - 6	0	Contents of Cond. Code Register
		10	0	Stack Pointer - 7	1	Irrelevant Data (Note 1)
		11	1	Vector Address FFFA (Hex)	1	Address of Subroutine (High Order Byte)
		12	1	Vector Address FFFB (Hex)	1	Address of Subroutine (Low Order Byte)
RELATIVE						
BCC BHI BNE BCS BLE BPL BEQ BLS BRA BGE BLT BVC BGT BMI BVS	4	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Branch Offset
		3	0	Op Code Address + 2	1	Irrelevant Data (Note 1)
		4	0	Branch Address	1	Irrelevant Data (Note 1)
BSR	8	1	1	Op Code Address	1	Op Code
		2	1	Op Code Address + 1	1	Branch Offset
		3	0	Return Address of Main Program	1	Irrelevant Data (Note 1)
		4	1	Stack Pointer	0	Return Address (Low Order Byte)
		5	1	Stack Pointer - 1	0	Return Address (High Order Byte)
		6	0	Stack Pointer - 2	1	Irrelevant Data (Note 1)
		7	0	Return Address of Main Program	1	Irrelevant Data (Note 1)
8	0	Subroutine Address	1	Irrelevant Data (Note 1)		

Note 1. If device which is addressed during this cycle uses VMA, then the Data Bus will go to the high impedance three-state condition. Depending on bus capacitance, data from the previous cycle may be retained on the Data Bus.

Note 2. Data is ignored by the MPU.

Note 3. For TST, VMA = 0 and Operand data does not change.

Note 4. While the MPU is waiting for the interrupt, Bus Available will go high indicating the following states of the control lines: VMA is low; Address Bus, R/W, and Data Bus are all in the high impedance state.


**MOTOROLA Semiconductor Products Inc.**


**MOTOROLA**  
**Semiconductors**

BOX 20912 • PHOENIX, ARIZONA 85036

### PERIPHERAL INTERFACE ADAPTER (PIA)

The MC6820 Peripheral Interface Adapter provides the universal means of interfacing peripheral equipment to the MC6800 Microprocessing Unit (MPU). This device is capable of interfacing the MPU to peripherals through two 8-bit bidirectional peripheral data buses and four control lines. No external logic is required for interfacing to most peripheral devices.

The functional configuration of the PIA is programmed by the MPU during system initialization. Each of the peripheral data lines can be programmed to act as an input or output, and each of the four control/interrupt lines may be programmed for one of several control modes. This allows a high degree of flexibility in the over-all operation of the interface.

- 8-Bit Bidirectional Data Bus for Communication with the MPU
- Two Bidirectional 8-Bit Buses for Interface to Peripherals
- Two Programmable Control Registers
- Two Programmable Data Direction Registers
- Four Individually-Controlled Interrupt Input Lines; Two Usable as Peripheral Control Outputs
- Handshake Control Logic for Input and Output Peripheral Operation
- High-Impedance 3-State and Direct Transistor Drive Peripheral Lines
- Program Controlled Interrupt and Interrupt Disable Capability
- CMOS Drive Capability on Side A Peripheral Lines

**MC6820**

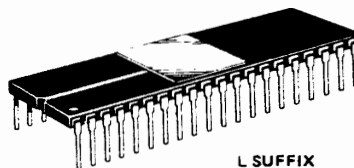
(0 to 70°C; L or P Suffix)

**MC6820C**

(-40 to 85°C; L Suffix only)

**MOS**

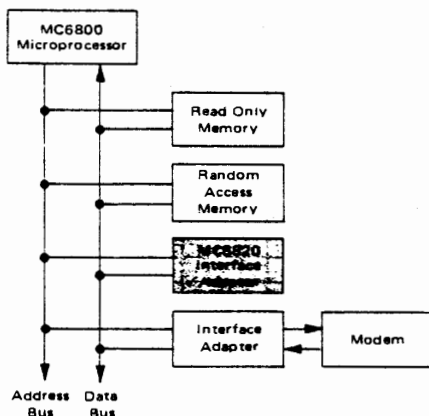
(N-CHANNEL, SILICON-GATE)

**PERIPHERAL INTERFACE  
ADAPTER**


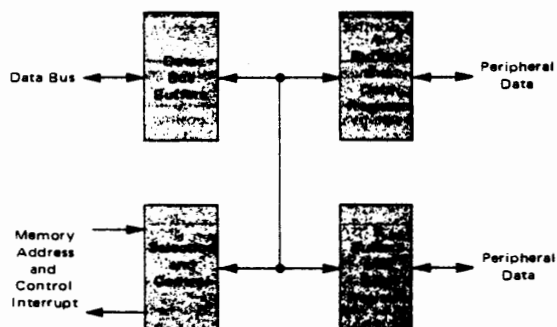
L SUFFIX  
CERAMIC PACKAGE  
CASE 715

NOT SHOWN: P SUFFIX  
PLASTIC PACKAGE  
CASE 711

#### M6800 MICROCOMPUTER FAMILY BLOCK DIAGRAM



#### MC6820 PERIPHERAL INTERFACE ADAPTER BLOCK DIAGRAM



## MC6820

ELECTRICAL CHARACTERISTICS ( $V_{CC} = 5.0 \text{ V} \pm 5\%$ ,  $V_{SS} = 0$ ,  $T_A = 0$  to  $70^\circ\text{C}$  unless otherwise noted.)

Characteristic	Symbol	Min	Typ	Max	Unit
Input High Voltage Enable Other Inputs	$V_{IH}$	$V_{SS} + 2.4$ $V_{SS} + 2.0$	— —	$V_{CC}$ $V_{CC}$	Vdc
Input Low Voltage Enable Other Inputs	$V_{IL}$	$V_{SS} - 0.3$ $V_{SS} - 0.3$	— —	$V_{SS} + 0.4$ $V_{SS} + 0.8$	Vdc
Input Leakage Current R/W, Reset, RS0, RS1, CS0, CS1, CS2, CA1, CB1, Enable ( $V_{in} = 0$ to $5.25 \text{ Vdc}$ )	$I_{in}$	—	1.0	2.5	$\mu\text{Adc}$
Three-State (Off State) Input Current D0-D7, PB0-PB7, CB2 ( $V_{in} = 0.4$ to $2.4 \text{ Vdc}$ )	$I_{TSI}$	—	2.0	10	$\mu\text{Adc}$
Input High Current PA0-PA7, CA2 ( $V_{IH} = 2.4 \text{ Vdc}$ )	$I_{IH}$	-100	-250	—	$\mu\text{Adc}$
Input Low Current PA0-PA7, CA2 ( $V_{IL} = 0.4 \text{ Vdc}$ )	$I_{IL}$	—	-1.0	-1.6	mAdc
Output High Voltage ( $I_{Load} = -205 \mu\text{Adc}$ , Enable Pulse Width < $25 \mu\text{s}$ ) ( $I_{Load} = -100 \mu\text{Adc}$ , Enable Pulse Width < $25 \mu\text{s}$ )	$V_{OH}$	$V_{SS} + 2.4$ $V_{SS} + 2.4$	— —	— —	Vdc
Output Low Voltage ( $I_{Load} = 1.6 \text{ mAdc}$ , Enable Pulse Width < $25 \mu\text{s}$ )	$V_{OL}$	—	—	$V_{SS} + 0.4$	Vdc
Output High Current (Sourcing) ( $V_{OH} = 2.4 \text{ Vdc}$ )	$I_{OH}$	-205 -100	— —	— —	$\mu\text{Adc}$ $\mu\text{Adc}$
( $V_O = 1.5 \text{ Vdc}$ , the current for driving other than TTL, e.g., Darlington Base)		-1.0	-2.5	-10	mAdc
Output Low Current (Sinking) ( $V_{OL} = 0.4 \text{ Vdc}$ )	$I_{OL}$	1.6	—	—	mAdc
Output Leakage Current (Off State) ( $V_{OH} = 2.4 \text{ Vdc}$ )	$I_{LOH}$	—	1.0	10	$\mu\text{Adc}$
Power Dissipation	$P_D$	—	—	650	mW
Input Capacitance ( $V_{in} = 0$ , $T_A = 25^\circ\text{C}$ , $f = 1.0 \text{ MHz}$ )	$C_{in}$	—	—	20 12.5 10 7.5	pF
Output Capacitance ( $V_{in} = 0$ , $T_A = 25^\circ\text{C}$ , $f = 1.0 \text{ MHz}$ )	$C_{out}$	—	—	5.0 10	pF
Peripheral Data Setup Time (Figure 1)	$t_{pDSU}$	200	—	—	ns
Delay Time, Enable negative transition to CA2 negative transition (Figure 2, 3)	$t_{CA2}$	—	—	1.0	$\mu\text{s}$
Delay Time, Enable negative transition to CA2 positive transition (Figure 2)	$t_{RS1}$	—	—	1.0	$\mu\text{s}$
Rise and Fall Times for CA1 and CA2 input signals (Figure 3)	$t_r, t_f$	—	—	1.0	$\mu\text{s}$
Delay Time from CA1 active transition to CA2 positive transition (Figure 3)	$t_{RS2}$	—	—	2.0	$\mu\text{s}$
Delay Time, Enable negative transition to Peripheral Data valid (Figures 4, 5)	$t_{PDW}$	—	—	1.0	$\mu\text{s}$
Delay Time, Enable negative transition to Peripheral CMOS Data Valid ( $V_{CC} = 30\% V_{CC}$ , Figure 4; Figure 12 Load C)	$t_{CMOS}$	—	—	2.0	$\mu\text{s}$
Delay Time, Enable positive transition to CB2 negative transition (Figure 6, 7)	$t_{CB2}$	—	—	1.0	$\mu\text{s}$
Delay Time, Peripheral Data valid to CB2 negative transition (Figure 5)	$t_{DC}$	20	—	—	ns
Delay Time, Enable positive transition to CB2 positive transition (Figure 6)	$t_{RS1}$	—	—	1.0	$\mu\text{s}$
Rise and Fall Time for CB1 and CB2 input signals (Figure 7)	$t_r, t_f$	—	—	1.0	$\mu\text{s}$
Delay Time, CB1 active transition to CB2 positive transition (Figure 7)	$t_{RS2}$	—	—	2.0	$\mu\text{s}$
Interrupt Release Time, IRQA and IRQB (Figure 8)	$t_{IR}$	—	—	1.6	$\mu\text{s}$
Reset Low Time* (Figure 9)	$t_{RL}$	2.0	—	—	$\mu\text{s}$

\*The Reset line must be high a minimum of  $1.0 \mu\text{s}$  before addressing the PIA.


**MOTOROLA Semiconductor Products Inc.**

**MC6820**
**MAXIMUM RATINGS**

Rating	Symbol	Value	Unit
Supply Voltage	$V_{CC}$	-0.3 to +7.0	Vdc
Input Voltage	$V_{in}$	-0.3 to +7.0	Vdc
Operating Temperature Range	$T_A$	0 to +70	°C
Storage Temperature Range	$T_{stg}$	-55 to +150	°C
Thermal Resistance	$\theta_{JA}$	82.5	°C/W

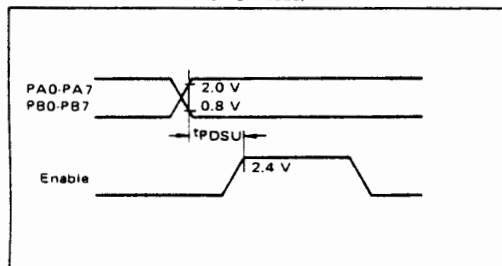
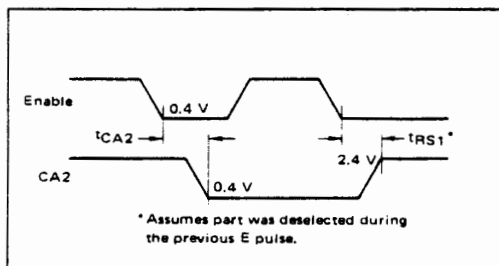
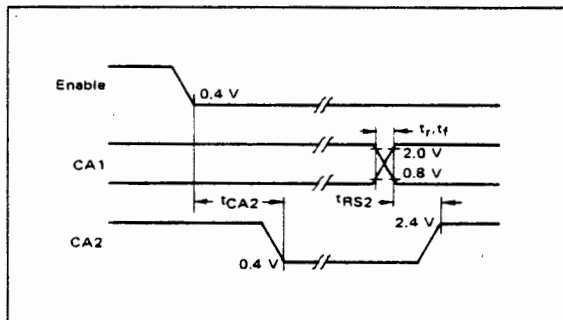
This device contains circuitry to protect the inputs against damage due to high static voltages or electric fields; however, it is advised that normal precautions be taken to avoid application of any voltage higher than maximum rated voltages to this high impedance circuit.

**BUS TIMING CHARACTERISTICS**
**READ** (Figures 10 and 12)

Characteristic	Symbol	Min	Typ	Max	Unit
Enable Cycle Time	$t_{cycE}$	1.0	—	—	$\mu s$
Enable Pulse Width, High	$PW_{EH}$	0.45	—	25	$\mu s$
Enable Pulse Width, Low	$PW_{EL}$	0.43	—	—	$\mu s$
Setup Time, Address and R/W valid to Enable positive transition	$t_{AS}$	160	—	—	ns
Data Delay Time	$t_{DDR}$	—	—	320	ns
Data Hold Time	$t_H$	10	—	—	ns
Address Hold Time	$t_{AH}$	10	—	—	ns
Rise and Fall Time for Enable input	$t_{Er}, t_{Ef}$	—	—	25	ns

**WRITE** (Figures 11 and 12)

Characteristic	Symbol	Min	Typ	Max	Unit
Enable Cycle Time	$t_{cycE}$	1.0	—	—	$\mu s$
Enable Pulse Width, High	$PW_{EH}$	0.45	—	25	$\mu s$
Enable Pulse Width, Low	$PW_{EL}$	0.43	—	—	$\mu s$
Setup Time, Address and R/W valid to Enable positive transition	$t_{AS}$	160	—	—	ns
Data Setup Time	$t_{DSW}$	195	—	—	ns
Data Hold Time	$t_H$	10	—	—	ns
Address Hold Time	$t_{AH}$	10	—	—	ns
Rise and Fall Time for Enable input	$t_{Er}, t_{Ef}$	—	—	25	ns

**FIGURE 1 — PERIPHERAL DATA SETUP TIME**  
 (Read Mode)

**FIGURE 2 — CA2 DELAY TIME**  
 (Read Mode; CRA-5 = CRA-3 = 1, CRA-4 = 0)

**FIGURE 3 — CA2 DELAY TIME**  
 (Read Mode; CRA-5 = 1, CRA-3 = CRA-4 = 0)

**MOTOROLA Semiconductor Products Inc.**

## MC6820

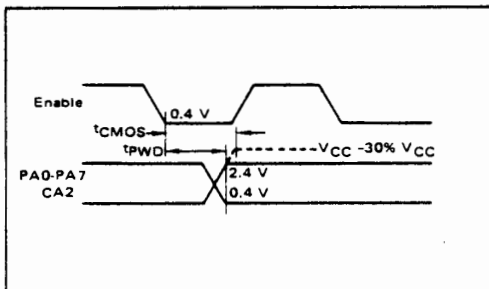
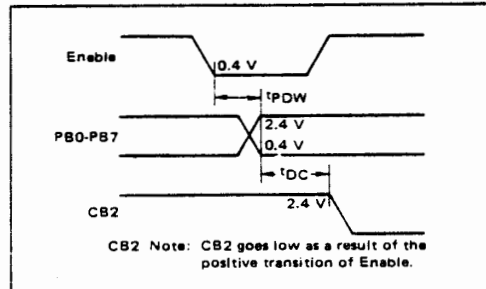
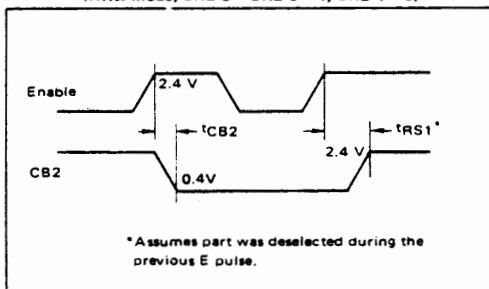
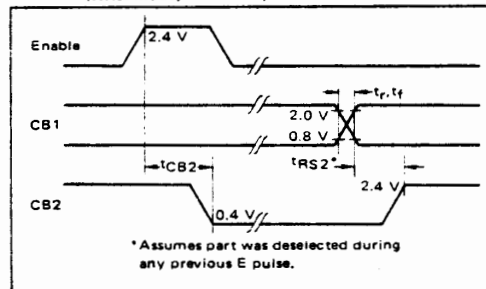
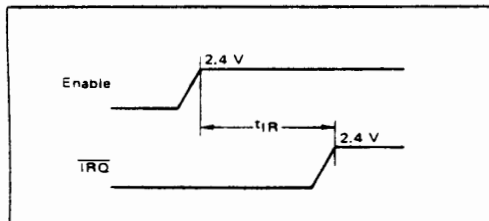
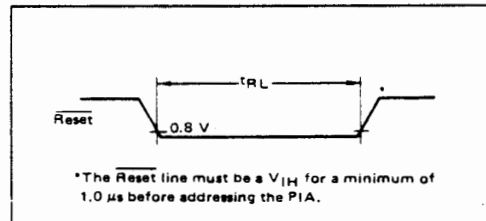
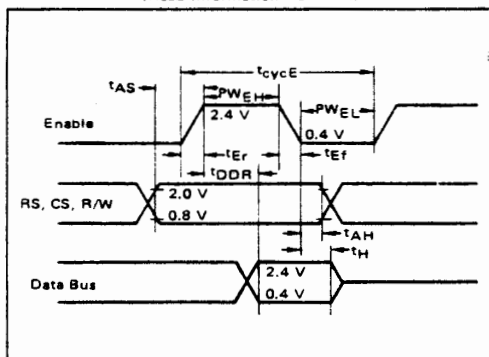
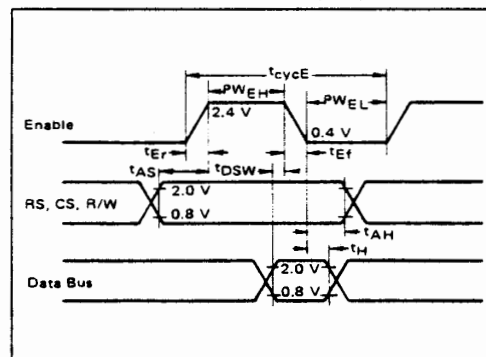
FIGURE 4 – PERIPHERAL CMOS DATA DELAY TIMES  
(Write Mode; CRA-5 = CRA-3 = 1, CRA-4 = 0)FIGURE 5 – PERIPHERAL DATA AND CB2 DELAY TIMES  
(Write Mode; CRB-5 = CRB-3 = 1, CRB-4 = 0)FIGURE 6 – CB2 DELAY TIME  
(Write Mode; CRB-5 = CRB-3 = 1, CRB-4 = 0)FIGURE 7 – CB2 DELAY TIME  
(Write Mode; CRB-5 = 1, CRB-3 = CRB-4 = 0)FIGURE 8 –  $\overline{IRQ}$  RELEASE TIME

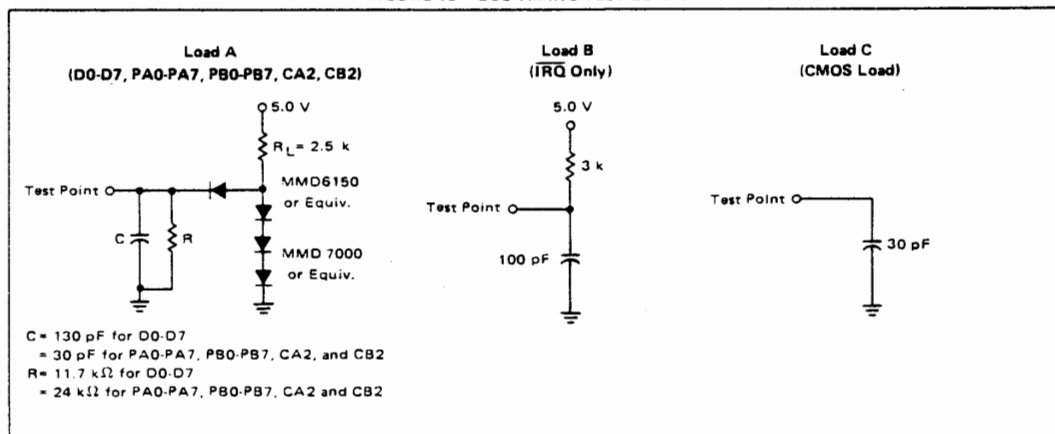
FIGURE 9 – RESET LOW TIME

FIGURE 10 – BUS READ TIMING CHARACTERISTICS  
(Read Information from PIA)FIGURE 11 – BUS WRITE TIMING CHARACTERISTICS  
(Write Information into PIA)



## MC6820

FIGURE 12 – BUS TIMING TEST LOADS



## PIA INTERFACE SIGNALS FOR MPU

The PIA interfaces to the MC6800 MPU with an eight-bit bi-directional data bus, three chip select lines, two register select lines, two interrupt request lines, read/write line, enable line and reset line. These signals, in conjunction with the MC6800 VMA output, permit the MPU to have complete control over the PIA. VMA should be utilized in conjunction with an MPU address line into a chip select of the PIA.

**PIA Bi-Directional Data (D0-D7)** – The bi-directional data lines (D0-D7) allow the transfer of data between the MPU and the PIA. The data bus output drivers are three-state devices that remain in the high-impedance (off) state except when the MPU performs a PIA read operation. The Read/Write line is in the Read (high) state when the PIA is selected for a Read operation.

**PIA Enable (E)** – The enable pulse, E, is the only timing signal that is supplied to the PIA. Timing of all other signals is referenced to the leading and trailing edges of the E pulse. This signal will normally be a derivative of the MC6800  $\phi_2$  Clock.

**PIA Read/Write (R/W)** – This signal is generated by the MPU to control the direction of data transfers on the Data Bus. A low state on the PIA Read/Write line enables the input buffers and data is transferred from the MPU to the PIA on the E signal if the device has been selected. A high on the Read/Write line sets up the PIA for a transfer of data to the bus. The PIA output buffers are enabled when the proper address and the enable pulse E are present.

**Reset** – The active low  $\overline{\text{Reset}}$  line is used to reset all register bits in the PIA to a logical zero (low). This line can be used as a power-on reset and as a master reset during system operation.

**PIA Chip Select ( $\overline{\text{CS0}}$ ,  $\overline{\text{CS1}}$  and  $\overline{\text{CS2}}$ )** – These three input signals are used to select the PIA.  $\overline{\text{CS0}}$  and  $\overline{\text{CS1}}$  must be high and  $\overline{\text{CS2}}$  must be low for selection of the device. Data transfers are then performed under the control of the Enable and Read/Write signals. The chip select lines must be stable for the duration of the E pulse. The device is deselected when any of the chip selects are in the inactive state.

**PIA Register Select ( $\overline{\text{RS0}}$  and  $\overline{\text{RS1}}$ )** – The two register select lines are used to select the various registers inside the PIA. These two lines are used in conjunction with internal Control Registers to select a particular register that is to be written or read.

The register and chip select lines should be stable for the duration of the E pulse while in the read or write cycle.

**Interrupt Request ( $\overline{\text{IRQA}}$  and  $\overline{\text{IRQB}}$ )** – The active low Interrupt Request lines ( $\overline{\text{IRQA}}$  and  $\overline{\text{IRQB}}$ ) act to interrupt the MPU either directly or through interrupt priority circuitry. These lines are "open drain" (no load device on the chip). This permits all interrupt request lines to be tied together in a wire-OR configuration.

Each Interrupt Request line has two internal interrupt flag bits that can cause the Interrupt Request line to go low. Each flag bit is associated with a particular peripheral interrupt line. Also four interrupt enable bits are provided in the PIA which may be used to inhibit a particular interrupt from a peripheral device.

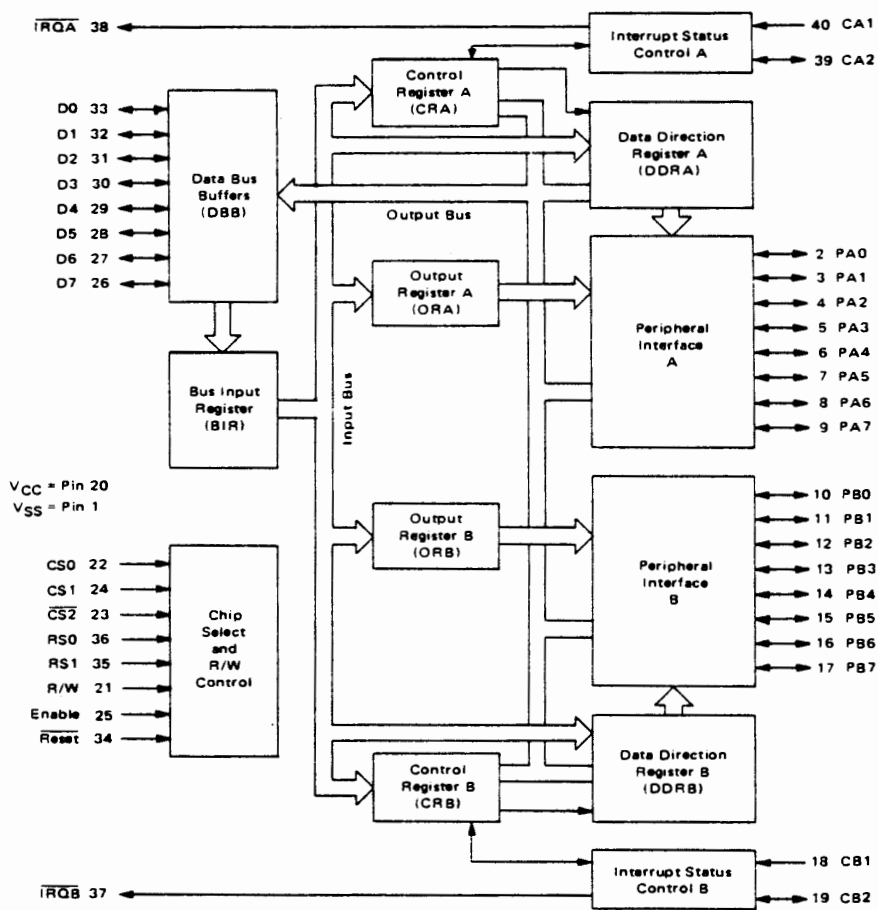
Servicing an interrupt by the MPU may be accomplished by a software routine that, on a prioritized basis, sequentially reads and tests the two control registers in each PIA for interrupt flag bits that are set.

The interrupt flags are cleared (zeroed) as a result of an



MC6820

## EXPANDED BLOCK DIAGRAM



MOTOROLA Semiconductor Products Inc.

## MC6820

MPU Read Peripheral Data Operation of the corresponding data register. After being cleared, the interrupt flag bit cannot be enabled to be set until the PIA is deselected during an E pulse. The E pulse is used to condition the interrupt control lines (CA1, CA2, CB1, CB2). When these lines are used as interrupt inputs at least one E

pulse must occur from the inactive edge to the active edge of the interrupt input signal to condition the edge sense network. If the interrupt flag has been enabled and the edge sense circuit has been properly conditioned, the interrupt flag will be set on the next active transition of the interrupt input pin.

## PIA PERIPHERAL INTERFACE LINES

The PIA provides two 8-bit bi-directional data buses and four interrupt/control lines for interfacing to peripheral devices.

**Section A Peripheral Data (PA0-PA7)** — Each of the peripheral data lines can be programmed to act as an input or output. This is accomplished by setting a "1" in the corresponding Data Direction Register bit for those lines which are to be outputs. A "0" in a bit of the Data Direction Register causes the corresponding peripheral data line to act as an input. During an MPU Read Peripheral Data Operation, the data on peripheral lines programmed to act as inputs appears directly on the corresponding MPU Data Bus lines. In the input mode the internal pullup resistor on these lines represents a maximum of one standard TTL load.

The data in Output Register A will appear on the data lines that are programmed to be outputs. A logical "1" written into the register will cause a "high" on the corresponding data line while a "0" results in a "low". Data in Output Register A may be read by an MPU "Read Peripheral Data A" operation when the corresponding lines are programmed as outputs. This data will be read properly if the voltage on the peripheral data lines is greater than 2.0 volts for a logic "1" output and less than 0.8 volt for a logic "0" output. Loading the output lines such that the voltage on these lines does not reach full voltage causes the data transferred into the MPU on a Read operation to differ from that contained in the respective bit of Output Register A.

**Section B Peripheral Data (PB0-PB7)** — The peripheral data lines in the B Section of the PIA can be programmed to act as either inputs or outputs in a similar manner to PA0-PA7. However, the output buffers driving these lines differ from those driving lines PA0-PA7. They have three-

state capability, allowing them to enter a high impedance state when the peripheral data line is used as an input. In addition, data on the peripheral data lines PB0-PB7 will be read properly from those lines programmed as outputs even if the voltages are below 2.0 volts for a "high". As outputs, these lines are compatible with standard TTL and may also be used as a source of up to 1 milliampere at 1.5 volts to directly drive the base of a transistor switch.

**Interrupt Input (CA1 and CB1)** — Peripheral input lines CA1 and CB1 are input only lines that set the interrupt flags of the control registers. The active transition for these signals is also programmed by the two control registers.

**Peripheral Control (CA2)** — The peripheral control line CA2 can be programmed to act as an interrupt input or as a peripheral control output. As an output, this line is compatible with standard TTL; as an input the internal pullup resistor on this line represents one standard TTL load. The function of this signal line is programmed with Control Register A.

**Peripheral Control (CB2)** — Peripheral Control line CB2 may also be programmed to act as an interrupt input or peripheral control output. As an input, this line has high input impedance and is compatible with standard TTL. As an output it is compatible with standard TTL and may also be used as a source of up to 1 milliampere at 1.5 volts to directly drive the base of a transistor switch. This line is programmed by Control Register B.

**NOTE:** It is recommended that the control lines (CA1, CA2, CB1, CB2) should be held in a logic 1 state when *Reset* is active to prevent setting of corresponding interrupt flags in the control register when *Reset* goes to an inactive state. Subsequent to *Reset* going inactive, a read of the data registers may be used to clear any undesired interrupt flags.



## MC6820

## INTERNAL CONTROLS

There are six locations within the PIA accessible to the MPU data bus: two Peripheral Registers, two Data Direction Registers, and two Control Registers. Selection of these locations is controlled by the RS0 and RS1 inputs together with bit 2 in the Control Register, as shown in Table 1.

TABLE 1 — INTERNAL ADDRESSING

RS1	RS0	Control Register Bit		Location Selected
		CRA-2	CRB-2	
0	0	1	X	Peripheral Register A
0	0	0	X	Data Direction Register A
0	1	X	X	Control Register A
1	0	X	1	Peripheral Register B
1	0	X	0	Data Direction Register B
1	1	X	X	Control Register B

X = Don't Care

## INITIALIZATION

A low reset line has the effect of zeroing all PIA registers. This will set PA0-PA7, PB0-PB7, CA2 and CB2 as inputs, and all interrupts disabled. The PIA must be configured during the restart program which follows the reset.

Details of possible configurations of the Data Direction and Control Register are as follows.

## DATA DIRECTION REGISTERS (DDRA and DDRB)

The two Data Direction Registers allow the MPU to control the direction of data through each corresponding peripheral data line. A Data Direction Register bit set at "0" configures the corresponding peripheral data line as an input; a "1" results in an output.

## CONTROL REGISTERS (CRA and CRB)

The two Control Registers (CRA and CRB) allow the MPU to control the operation of the four peripheral control lines CA1, CA2, CB1 and CB2. In addition they allow the MPU to enable the interrupt lines and monitor the status of the interrupt flags. Bits 0 through 5 of the two registers may be written or read by the MPU when the proper chip select and register select signals are applied. Bits 6 and 7 of the two registers are read only and are modified by external interrupts occurring on control lines CA1, CA2, CB1 or CB2. The format of the control words is shown in Table 2.

TABLE 2 — CONTROL WORD FORMAT

	7	6	5	4	3	2	1	0
CRA	IRQA1	IRQA2	CA2 Control			DDRA Access	CA1 Control	
CRB	IRQB1	IRQB2	CB2 Control			DDRB Access	CB1 Control	

## Data Direction Access Control Bit (CRA-2 and CRB-2) —

Bit 2 in each Control register (CRA and CRB) allows selection of either a Peripheral Interface Register or the Data Direction Register when the proper register select signals are applied to RS0 and RS1.

## Interrupt Flags (CRA-6, CRA-7, CRB-6, and CRB-7) —

The four interrupt flag bits are set by active transitions of signals on the four Interrupt and Peripheral Control lines when those lines are programmed to be inputs. These bits cannot be set directly from the MPU Data Bus and are reset indirectly by a Read Peripheral Data Operation on the appropriate section.

TABLE 3 — CONTROL OF INTERRUPT INPUTS CA1 AND CB1

CRA-1 (CRB-1)	CRA-0 (CRB-0)	Interrupt Input CA1 (CB1)	Interrupt Flag CRA-7 (CRB-7)	MPU Interrupt Request IRQA (IRQB)
0	0	↓ Active	Set high on ↓ of CA1 (CB1)	Disabled — IRQ remains high
0	1	↓ Active	Set high on ↓ of CA1 (CB1)	Goes low when the interrupt flag bit CRA-7 (CRB-7) goes high
1	0	↑ Active	Set high on ↑ of CA1 (CB1)	Disabled — IRQ remains high
1	1	↑ Active	Set high on ↑ of CA1 (CB1)	Goes low when the interrupt flag bit CRA-7 (CRB-7) goes high

- Notes:
- ↑ indicates positive transition (low to high)
  - ↓ indicates negative transition (high to low)
  - The Interrupt flag bit CRA-7 is cleared by an MPU Read of the A Data Register, and CRB-7 is cleared by an MPU Read of the B Data Register.
  - If CRA-0 (CRB-0) is low when an interrupt occurs (Interrupt disabled) and is later brought high, IRQA (IRQB) occurs after CRA-0 (CRB-0) is written to a "one".


**MOTOROLA Semiconductor Products Inc.**

**MC6820**

Control of CA1 and CB1 Interrupt Input Lines (CRA-0, CRB-0, CRA-1, and CRB-1) — The two lowest order bits of the control registers are used to control the interrupt input lines CA1 and CB1. Bits CRA-0 and CRB-0 are

used to enable the MPU interrupt signals  $\overline{IROA}$  and  $\overline{IROB}$ , respectively. Bits CRA-1 and CRB-1 determine the active transition of the interrupt input signals CA1 and CB1 (Table 3).

**TABLE 4 — CONTROL OF CA2 AND CB2 AS INTERRUPT INPUTS**  
 CRA5 (CRB5) is low

CRA-5 (CRB-5)	CRA-4 (CRB-4)	CRA-3 (CRB-3)	Interrupt Input CA2 (CB2)	Interrupt Flag CRA-6 (CRB-6)	MPU Interrupt Request $\overline{IROA}$ ( $\overline{IROB}$ )
0	0	0	↓ Active	Set high on ↓ of CA2 (CB2)	Disabled — $\overline{IRO}$ re- mains high
0	0	1	↓ Active	Set high on ↓ of CA2 (CB2)	Goes low when the interrupt flag bit CRA-6 (CRB-6) goes high
0	1	0	* Active	Set high on ↑ of CA2 (CB2)	Disabled — $\overline{IRO}$ re- mains high
0	1	1	* Active	Set high on ↑ of CA2 (CB2)	Goes low when the interrupt flag bit CRA-6 (CRB-6) goes high

- Notes: 1. \* indicates positive transition (low to high)  
 2. ↓ indicates negative transition (high to low)  
 3. The interrupt flag bit CRA-6 is cleared by an MPU Read of the A Data Register and CRB-6 is cleared by an MPU Read of the B Data Register.  
 4. If CRA-3 (CRB-3) is low when an interrupt occurs (Interrupt disabled) and is later brought high,  $\overline{IROA}$  ( $\overline{IROB}$ ) occurs after CRA-3 (CRB-3) is written to a "one".

**TABLE 5 — CONTROL OF CB2 AS AN OUTPUT**  
 CRB-5 is high

CRB-5	CRB-4	CRB-3	CB2	
			Cleared	Set
1	0	0	Low on the positive transition of the first E pulse following an MPU Write "B" Data Register operation.	High when the interrupt flag bit CRB-7 is set by an active transition of the CB1 signal.
1	0	1	Low on the positive transition of the first E pulse after an MPU Write "B" Data Register operation.	High on the positive edge of the first "E" pulse following an "E" pulse which occurred while the part was deselected.
1	1	0	Low when CRB-3 goes low as a result of an MPU Write in Control Register "B".	Always low as long as CRB-3 is low. Will go high on an MPU Write in Control Register "B" that changes CRB-3 to "one".
1	1	1	Always high as long as CRB-3 is high. Will be cleared when an MPU Write Control Register "B" results in clearing CRB-3 to "zero".	High when CRB-3 goes high as a result of an MPU Write into Control Register "B".



## MC6820

**Control of CA2 and CB2 Peripheral Control Lines (CRA-3, CRA-4, CRA-5, CRB-3, CRB-4, and CRB-5) —** Bits 3, 4, and 5 of the two control registers are used to control the CA2 and CB2 Peripheral Control lines. These bits determine if the control lines will be an interrupt input or an output control signal. If bit CRA-5 (CRB-5)

is low, CA2 (CB2) is an interrupt input line similar to CA1 (CB1) (Table 4). When CRA-5 (CRB-5) is high, CA2 (CB2) becomes an output signal that may be used to control peripheral data transfers. When in the output mode, CA2 and CB2 have slightly different characteristics (Tables 5 and 6).

**TABLE 6 — CONTROL OF CA-2 AS AN OUTPUT**  
CRA-5 is high

CRA-5	CRA-4	CRA-3	CA2	
			Cleared	Set
1	0	0	Low on negative transition of E after an MPU Read "A" Data operation.	High when the interrupt flag bit CRA-7 is set by an active transition of the CA1 signal.
1	0	1	Low on negative transition of E after an MPU Read "A" Data operation.	High on the negative edge of the first "E" pulse which occurs during a deselect.
1	1	0	Low when CRA-3 goes low as a result of an MPU Write to Control Register "A".	Always low as long as CRA-3 is low. Will go high on an MPU Write to Control Register "A" that changes CRA-3 to "one".
1	1	1	Always high as long as CRA-3 is high. Will be cleared on an MPU Write to Control Register "A" that clears CRA-3 to a "zero".	High when CRA-3 goes high as a result of an MPU Write to Control Register "A".

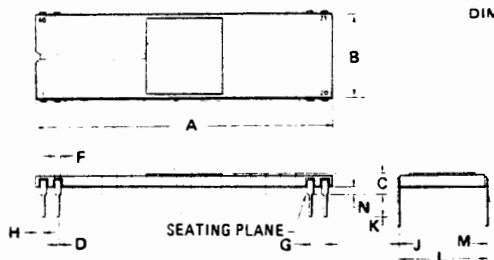
## PIN ASSIGNMENT

1	CA1	40
2	CA2	39
3	IRQA	38
4	IRQB	37
5	RS0	36
6	RS1	35
7	Reset	34
8	D0	33
9	D1	32
10	D2	31
11	D3	30
12	D4	29
13	D5	28
14	D6	27
15	D7	26
16	E	25
17	CS1	24
18	CS2	23
19	CS0	22
20	R/W	21

## PACKAGE DIMENSIONS

**CASE 715-02**  
 (CERAMIC)

SEE PAGE 165 FOR  
PLASTIC PACKAGE  
DIMENSIONS.



DIM	MILLIMETERS		INCHES	
	MIN	MAX	MIN	MAX
A	50.29	51.31	1.980	2.020
B	14.86	15.62	0.585	0.615
C	2.54	4.19	0.100	0.165
D	0.38	0.53	0.015	0.021
F	0.76	1.40	0.030	0.055
G	2.54 BSC		0.100 BSC	
H	0.76	1.78	0.030	0.070
J	0.20	0.33	0.008	0.013
K	2.54	4.19	0.100	0.165
L	14.60	15.37	0.575	0.605
M	—	10 <sup>0</sup>	—	10 <sup>0</sup>
N	0.51	1.52	0.020	0.060

## NOTE:

1. LEADS, TRUE POSITIONED WITHIN 0.25 mm (0.010) DIA (AT SEATING PLANE), AT MAX. MAT'L CONDITION.



**MOTOROLA Semiconductor Products Inc.**



**MOTOROLA**  
**Semiconductors**

BOX 20912 • PHOENIX, ARIZONA 85036

### ASYNCHRONOUS COMMUNICATIONS INTERFACE ADAPTER (ACIA)

The MC6850 Asynchronous Communications Interface Adapter provides the data formatting and control to interface serial asynchronous data communications information to bus organized systems such as the MC6800 Microprocessing Unit.

The bus interface of the MC6850 includes select, enable, read/write, interrupt and bus interface logic to allow data transfer over an 8-bit bi-directional data bus. The parallel data of the bus system is serially transmitted and received by the asynchronous data interface, with proper formatting and error checking. The functional configuration of the ACIA is programmed via the data bus during system initialization. A programmable Control Register provides variable word lengths, clock division ratios, transmit control, receive control, and interrupt control. For peripheral or modem operation three control lines are provided. These lines allow the ACIA to interface directly with the MC6860L 0-600 bps digital modem.

- Eight and Nine-Bit Transmission
- Optional Even and Odd Parity
- Parity, Overrun and Framing Error Checking
- Programmable Control Register
- Optional  $\div 1$ ,  $\div 16$ , and  $\div 64$  Clock Modes
- Up to 500 kbps Transmission
- False Start Bit Deletion
- Peripheral/Modem Control Functions
- Double Buffered
- One or Two Stop Bit Operation

**MC6850**

(0 to 70°C; L or P Suffix)

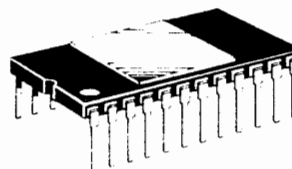
**MC6850C**

(-40 to 85°C; L Suffix only)

**MOS**

(N-CHANNEL, SILICON-GATE)

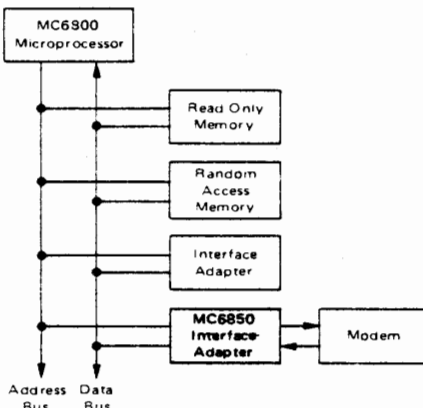
### ASYNCHRONOUS COMMUNICATIONS INTERFACE ADAPTER



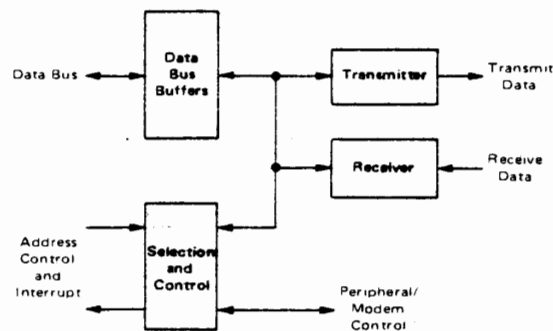
**L SUFFIX**  
CERAMIC PACKAGE  
CASE 716

NOT SHOWN: **P SUFFIX**  
PLASTIC PACKAGE  
CASE 709

**M6800 MICROCOMPUTER FAMILY  
BLOCK DIAGRAM**



**MC6850 ASYNCHRONOUS COMMUNICATIONS INTERFACE ADAPTER  
BLOCK DIAGRAM**



**MAXIMUM RATINGS**

Rating	Symbol	Value	Unit
Supply Voltage	V <sub>CC</sub>	-0.3 to +7.0	Vdc
Input Voltage	V <sub>in</sub>	-0.3 to +7.0	Vdc
Operating Temperature Range	T <sub>A</sub>	0 to +70	°C
Storage Temperature Range	T <sub>stg</sub>	-55 to +150	°C
Thermal Resistance	θ <sub>JA</sub>	82.5	°C/W

This device contains circuitry to protect the inputs against damage due to high static voltages or electric fields; however, it is advised that normal precautions be taken to avoid application of any voltage higher than maximum rated voltages to this high-impedance circuit.

**ELECTRICAL CHARACTERISTICS** (V<sub>CC</sub> = 5.0 V ±5%, V<sub>SS</sub> = 0, T<sub>A</sub> = 0 to 70°C unless otherwise noted.)

Characteristic	Symbol	Min	Typ	Max	Unit
Input High Voltage	V <sub>IH</sub>	V <sub>SS</sub> + 2.0	—	V <sub>CC</sub>	Vdc
Input Low Voltage	V <sub>IL</sub>	V <sub>SS</sub> - 0.3	—	V <sub>SS</sub> + 0.8	Vdc
Input Leakage Current (V <sub>in</sub> = 0 to 5.25 Vdc)	I <sub>in</sub>	—	1.0	2.5	μAdc
Three-State (Off State) Input Current (V <sub>in</sub> = 0.4 to 2.4 Vdc)	I <sub>TSI</sub>	—	2.0	10	μAdc
Output High Voltage (I <sub>Load</sub> = -205 μAdc, Enable Pulse Width < 25 μs) (I <sub>Load</sub> = -100 μAdc, Enable Pulse Width < 25 μs)	V <sub>OH</sub>	V <sub>SS</sub> + 2.4 V <sub>SS</sub> + 2.4	— —	— —	Vdc
Output Low Voltage (I <sub>Load</sub> = 1.6 mAdc, Enable Pulse Width < 25 μs)	V <sub>OL</sub>	—	—	V <sub>SS</sub> + 0.4	Vdc
Output Leakage Current (Off State) (V <sub>OH</sub> = 2.4 Vdc)	I <sub>LOH</sub>	—	1.0	10	μAdc
Power Dissipation	P <sub>D</sub>	—	300	525	mW
Input Capacitance (V <sub>in</sub> = 0, T <sub>A</sub> = 25°C, f = 1.0 MHz)	C <sub>in</sub>	—	10	12.5	pF
Output Capacitance (V <sub>in</sub> = 0, T <sub>A</sub> = 25°C, f = 1.0 MHz)	C <sub>out</sub>	—	—	10	pF
Minimum Clock Pulse Width, Low (Figure 1)	PW <sub>CL</sub>	600	—	—	ns
Minimum Clock Pulse Width, High (Figure 2)	PW <sub>CH</sub>	600	—	—	ns
Clock Frequency	f <sub>C</sub>	—	—	500	kHz
Clock-to-Data Delay for Transmitter (Figure 3)	t <sub>TDD</sub>	—	—	1.0	μs
Receive Data Setup Time (Figure 4)	t <sub>RDSU</sub>	500	—	—	ns
Receive Data Hold Time (Figure 5)	t <sub>RDH</sub>	500	—	—	ns
Interrupt Request Release Time (Figure 6)	t <sub>IR</sub>	—	—	1.2	μs
Request-to-Send Delay Time (Figure 6)	t <sub>RTS</sub>	—	—	1.0	μs
Input Transition Times (Except Enable)	t <sub>r</sub> , t <sub>f</sub>	—	—	1.0*	μs

\* 1.0 μs or 10% of the pulse width, whichever is smaller.

**BUS TIMING CHARACTERISTICS**
**READ** (Figures 7 and 9)

Characteristic	Symbol	Min	Typ	Max	Unit
Enable Cycle Time	t <sub>cycE</sub>	1.0	—	—	μs
Enable Pulse Width, High	PW <sub>EH</sub>	0.45	—	25	μs
Enable Pulse Width, Low	PW <sub>EL</sub>	0.43	—	—	μs
Setup Time, Address and R/W valid to Enable positive transition	t <sub>AS</sub>	160	—	—	ns
Data Delay Time	t <sub>DDR</sub>	—	—	320	ns
Data Hold Time	t <sub>H</sub>	10	—	—	ns
Address Hold Time	t <sub>AH</sub>	10	—	—	ns
Rise and Fall Time for Enable input	t <sub>Er</sub> , t <sub>Ef</sub>	—	—	25	ns

**WRITE** (Figure 8 and 9)

Characteristic	Symbol	Min	Typ	Max	Unit
Enable Cycle Time	t <sub>cycE</sub>	1.0	—	—	μs
Enable Pulse Width, High	PW <sub>EH</sub>	0.45	—	25	μs
Enable Pulse Width, Low	PW <sub>EL</sub>	0.43	—	—	μs
Setup Time, Address and R/W valid to Enable positive transition	t <sub>AS</sub>	160	—	—	ns
Data Setup Time	t <sub>DSW</sub>	195	—	—	ns
Data Hold Time	t <sub>H</sub>	10	—	—	ns
Address Hold Time	t <sub>AH</sub>	10	—	—	ns
Rise and Fall Time for Enable input	t <sub>Er</sub> , t <sub>Ef</sub>	—	—	25	ns


**MOTOROLA Semiconductor Products Inc.**



MC6850

FIGURE 1 – CLOCK PULSE WIDTH, LOW-STATE

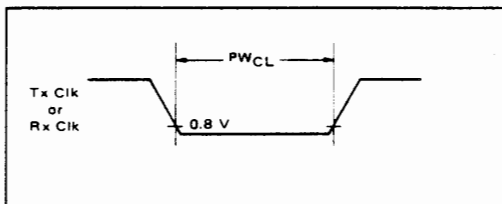


FIGURE 2 – CLOCK PULSE WIDTH, HIGH-STATE

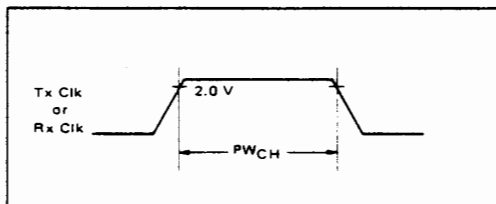


FIGURE 3 – TRANSMIT DATA OUTPUT DELAY

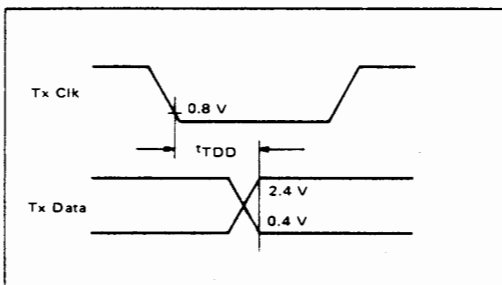


FIGURE 4 – RECEIVE DATA SETUP TIME  
(÷1 Mode)

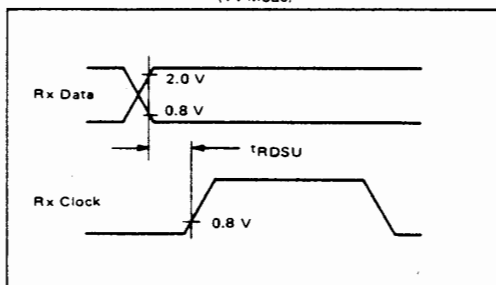


FIGURE 5 – RECEIVE DATA HOLD TIME  
(÷1 Mode)

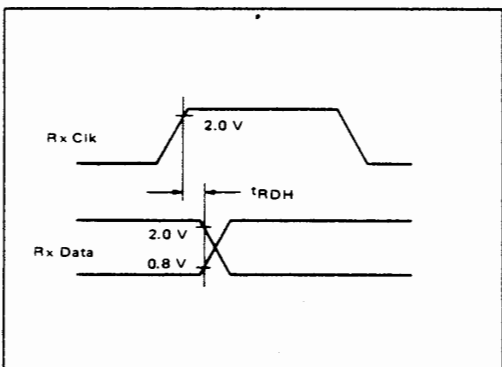


FIGURE 6 – REQUEST-TO-SEND DELAY AND  
INTERRUPT-REQUEST RELEASE TIMES

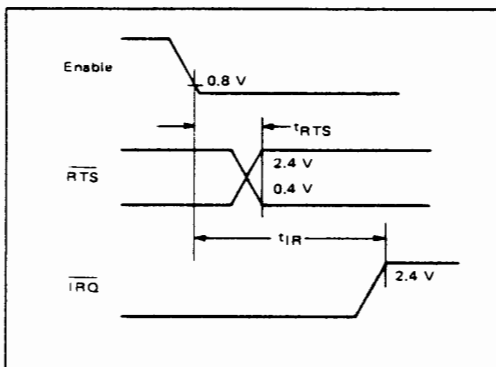


FIGURE 7 – BUS READ TIMING CHARACTERISTICS  
(Read information from ACIA)

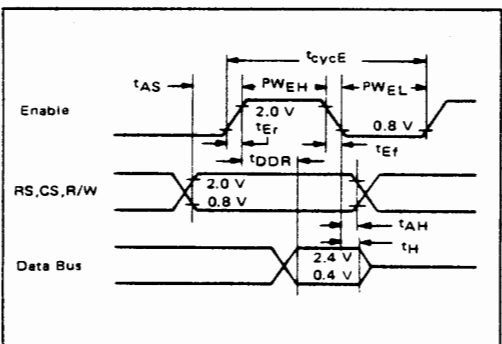
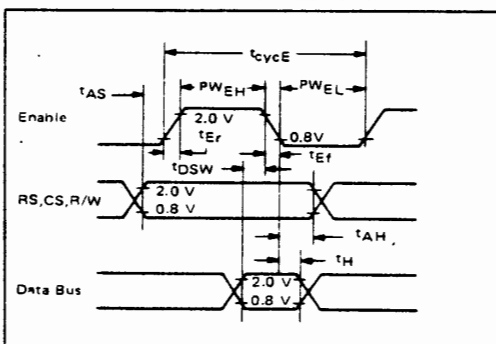
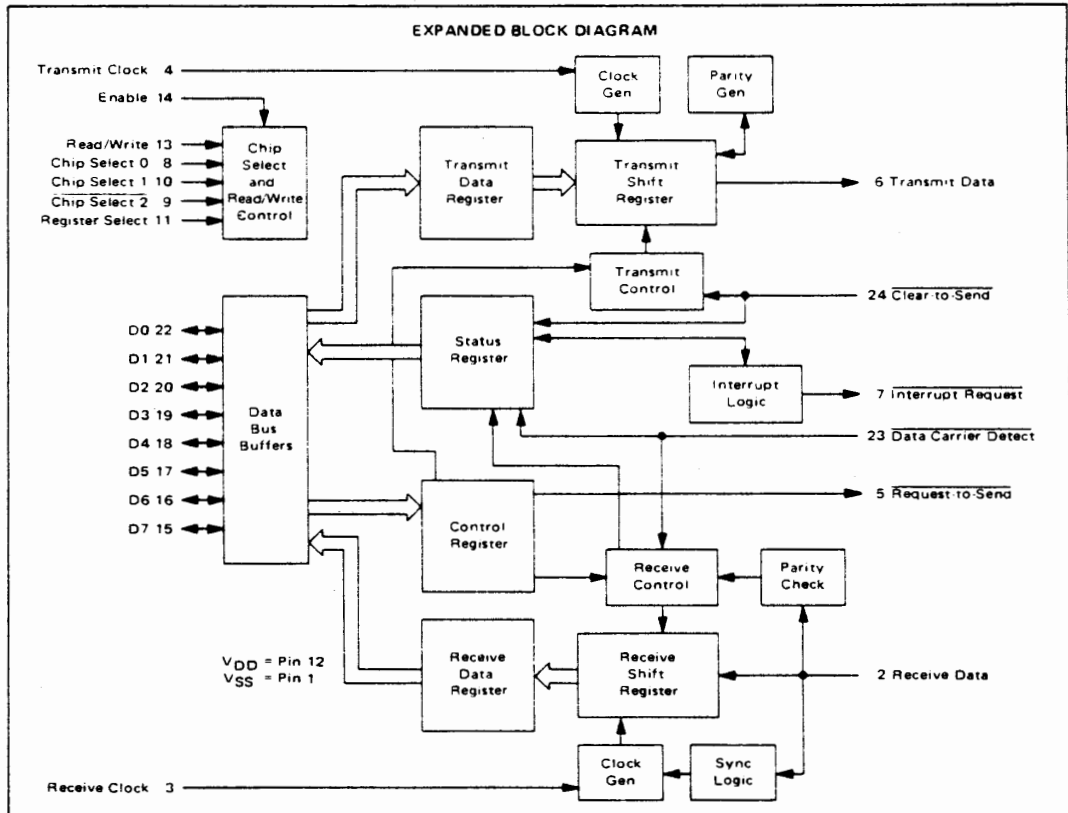
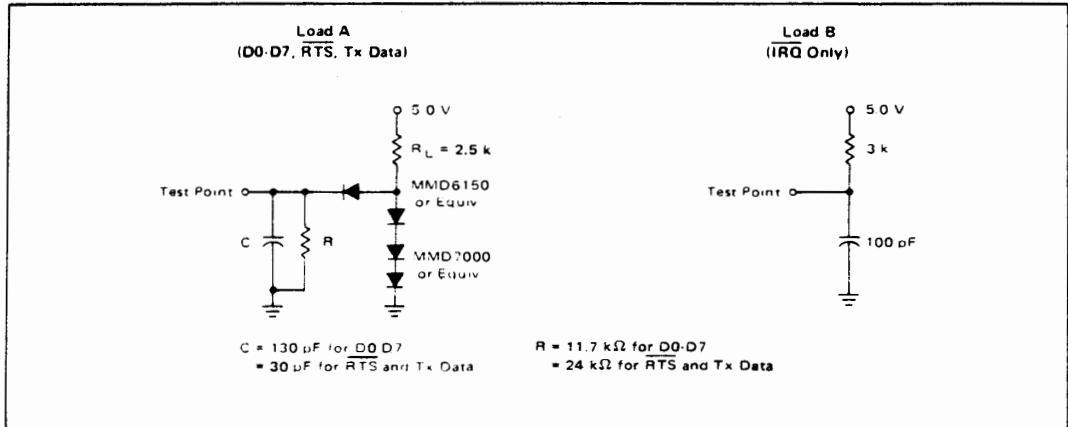


FIGURE 8 – BUS WRITE TIMING CHARACTERISTICS  
(Write information into ACIA)



## MC6850

FIGURE 9 — BUS TIMING TEST LOADS



## DEVICE OPERATION

At the bus interface, the ACIA appears as two addressable memory locations. Internally, there are four registers: two read-only and two write-only registers. The read-only

registers are Status and Receive Data; the write-only registers are Control and Transmit Data. The serial interface consists of serial input and output lines with independent clocks, and three peripheral/modem control lines.


**MOTOROLA Semiconductor Products Inc.**

**MC6850**
**POWER ON/MASTER RESET**

The master reset (CR0, CR1) should be set during system initialization to insure the reset condition and prepare for programming the ACIA functional configuration when the communications channel is required. Control bits CR5 and CR6 should also be programmed to define the state of RTS whenever master reset is utilized. The ACIA also contains internal power-on reset logic to detect the power line turn-on transition and hold the chip in a reset state to prevent erroneous output transitions prior to initialization. This circuitry depends on clean power turn-on transitions. The power-on reset is released by means of the bus-programmed master reset which must be applied prior to operating the ACIA. After master resetting the ACIA, the programmable Control Register can be set for a number of options such as variable clock divider ratios, variable word length, one or two stop bits, parity (even, odd, or none), etc.

**TRANSMIT**

A typical transmitting sequence consists of reading the ACIA Status Register either as a result of an interrupt or in the ACIA's turn in a polling sequence. A character may be written into the Transmit Data Register if the status read operation has indicated that the Transmit Data Register is empty. This character is transferred to a Shift Register where it is serialized and transmitted from the Transmit Data output preceded by a start bit and followed by one or two stop bits. Internal parity (odd or even) can be optionally added to the character and will occur between the last data bit and the first stop bit. After the first character is written in the Data Register, the Status Register can be read again to check for a Transmit Data Register Empty condition and current peripheral status. If the register is empty, another character can be loaded for transmission even though the first character is in the process of being transmitted (because of double buffering). The second character will be automatically transferred into the Shift Register when the first character transmission is completed. This sequence continues until all the characters have been transmitted.

**RECEIVE**

Data is received from a peripheral by means of the Receive Data input. A divide-by-one clock ratio is provided for an externally synchronized clock (to its data) while the divide-by-16 and 64 ratios are provided for internal synchronization. Bit synchronization in the divide-by-16 and 64 modes is initiated by the detection of the leading mark-to-space transition of the start bit. False start bit deletion capability insures that a full half bit of a start bit has been received before the internal clock is synchronized to the bit time. As a character is being received, parity (odd or even) will be checked and the error indication will be available in the Status Register along with framing error, overrun error, and Receive Data Register full. In a typical receiving sequence, the Status Register is read to determine if a character has been re-

ceived from a peripheral. If the Receiver Data Register is full, the character is placed on the 8-bit ACIA bus when a Read Data command is received from the MPU. When parity has been selected for an 8-bit word (7 bits plus parity), the receiver strips the parity bit (D7 = 0) so that data alone is transferred to the MPU. This feature reduces MPU programming. The Status Register can continue to be read again to determine when another character is available in the Receive Data Register. The receiver is also double buffered so that a character can be read from the data register as another character is being received in the shift register. The above sequence continues until all characters have been received.

**INPUT/OUTPUT FUNCTIONS**
**ACIA INTERFACE SIGNALS FOR MPU**

The ACIA interfaces to the MC6800 MPU with an 8-bit bi-directional data bus, three chip select lines, a register select line, an interrupt request line, read/write line, and enable line. These signals, in conjunction with the MC6800 VMA output, permit the MPU to have complete control over the ACIA.

**ACIA Bi-Directional Data (D0-D7)** — The bi-directional data lines (D0-D7) allow for data transfer between the ACIA and the MPU. The data bus output drivers are three-state devices that remain in the high-impedance (off) state except when the MPU performs an ACIA read operation.

**ACIA Enable (E)** — The Enable signal, E, is a high impedance TTL compatible input that enables the bus input/output data buffers and clocks data to and from the ACIA. This signal will normally be a derivative of the MC6800  $\phi 2$  Clock.

**Read/Write (R/W)** — The Read/Write line is a high impedance input that is TTL compatible and is used to control the direction of data flow through the ACIA's input/output data bus interface. When Read/Write is high (MPU Read cycle), ACIA output drivers are turned on and a selected register is read. When it is low, the ACIA output drivers are turned off and the MPU writes into a selected register. Therefore, the Read/Write signal is used to select read-only or write-only registers within the ACIA.

**Chip Select (CS0, CS1,  $\overline{CS2}$ )** — These three high impedance TTL compatible input lines are used to address the ACIA. The ACIA is selected when CS0 and CS1 are high and  $\overline{CS2}$  is low. Transfers of data to and from the ACIA are then performed under the control of the Enable signal, Read/Write, and Register Select.

**Register Select (RS)** — The Register Select line is a high impedance input that is TTL compatible. A high level is used to select the Transmit/Receive Data Registers and a low level the Control/Status Registers. The Read/Write signal line is used in conjunction with Register Select to select the read-only or write-only register in each register pair.

**Interrupt Request (IRQ)** — Interrupt Request is a TTL compatible, open-drain (no internal pullup), active low



**MC6850**

output that is used to interrupt the MPU. The  $\overline{\text{IRQ}}$  output remains low as long as the cause of the interrupt is present and the appropriate interrupt enable within the ACIA is set. The  $\overline{\text{IRQ}}$  status bit, when high, indicates the  $\overline{\text{IRQ}}$  output is in the active state.

Interrupts result from conditions in both the transmitter and receiver sections of the ACIA. The transmitter section causes an interrupt when the Transmitter Interrupt Enabled condition is selected (CR5 = CR6), and the Transmit Data Register Empty (TDRE) status bit is high. The TDRE status bit indicates the current status of the Transmitter Data Register except when inhibited by Clear-to-Send (CTS) being high or the ACIA being maintained in the Reset condition. The interrupt is cleared by writing data into the Transmit Data Register. The interrupt is masked by disabling the Transmitter Interrupt via CR5 or CR6 or by the loss of CTS which inhibits the TDRE status bit. The Receiver section causes an interrupt when the Receiver Interrupt Enable is set and the Receive Data Register Full (RDRF) status bit is high, an Overrun has occurred, or Data Carrier Detect (DCD) has gone high. An interrupt resulting from the RDRF status bit can be cleared by reading data or resetting the ACIA. Interrupts caused by Overrun or loss of DCD are cleared by reading the status register after the error condition has occurred and then reading the Receive Data Register or resetting the ACIA. The receiver interrupt is masked by resetting the Receiver Interrupt Enable.

**CLOCK INPUTS**

Separate high impedance TTL compatible inputs are provided for clocking of transmitted and received data. Clock frequencies of 1, 16 or 64 times the data rate may be selected.

**Transmit Clock (Tx Clk)** — The Transmit Clock input is used for the clocking of transmitted data. The transmitter initiates data on the negative transition of the clock.

**Receive Clock (Rx Clk)** — The Receive Clock input is used for synchronization of received data. (In the  $\div 1$  mode, the clock and data must be synchronized externally.) The receiver samples the data on the positive transition of the clock.

**SERIAL INPUT/OUTPUT LINES**

**Receive Data (Rx Data)** — The Receive Data line is a high impedance TTL compatible input through which data is received in a serial format. Synchronization with a clock for detection of data is accomplished internally when clock rates of 16 or 64 times the bit rate are used. Data rates are in the range of 0 to 500 kbps when external synchronization is utilized.

**Transmit Data (Tx Data)** — The Transmit Data output line transfers serial data to a modem or other peripheral. Data rates are in the range of 0 to 500 kbps when external synchronization is utilized.

**PERIPHERAL/MODEM CONTROL**

The ACIA includes several functions that permit limited

control of a peripheral or modem. The functions included are Clear-to-Send, Request-to-Send and Data Carrier Detect.

**Clear-to-Send (CTS)** — This high impedance TTL compatible input provides automatic control of the transmitting end of a communications link via the modem Clear-to-Send active low output by inhibiting the Transmit Data Register Empty (TDRE) status bit.

**Request-to-Send (RTS)** — The Request-to-Send output enables the MPU to control a peripheral or modem via the data bus. The RTS output corresponds to the state of the Control Register bits CR5 and CR6. When CR6 = 0 or both CR5 and CR6 = 1, the RTS output is low (the active state). This output can also be used for Data Terminal Ready (DTR).

**Data Carrier Detect (DCD)** — This high impedance TTL compatible input provides automatic control, such as in the receiving end of a communications link by means of a modem Data Carrier Detect output. The DCD input inhibits and initializes the receiver section of the ACIA when high. A low to high transition of the Data Carrier Detect initiates an interrupt to the MPU to indicate the occurrence of a loss of carrier when the Receive Interrupt Enable bit is set.

**ACIA REGISTERS**

The expanded block diagram for the ACIA indicates the internal registers on the chip that are used for the status, control, receiving, and transmitting of data. The content of each of the registers is summarized in Table 1.

**TRANSMIT DATA REGISTER (TDR)**

Data is written in the Transmit Data Register during the negative transition of the enable (E) when the ACIA has been addressed and RS = R/W is selected. Writing data into the register causes the Transmit Data Register Empty bit in the Status Register to go low. Data can then be transmitted. If the transmitter is idling and no character is being transmitted, then the transfer will take place within one bit time of the trailing edge of the Write command. If a character is being transmitted, the new data character will commence as soon as the previous character is complete. The transfer of data causes the Transmit Data Register Empty (TDRE) bit to indicate empty.

**RECEIVE DATA REGISTER (RDR)**

Data is automatically transferred to the empty Receive Data Register (RDR) from the receiver deserializer (a shift register) upon receiving a complete character. This event causes the Receive Data Register Full bit (RDRF) in the status buffer to go high (full). Data may then be read through the bus by addressing the ACIA and selecting the Receive Data Register with RS and R/W high when the ACIA is enabled. The non-destructive read cycle causes the RDRF bit to be cleared to empty although


**MOTOROLA Semiconductor Products Inc.**

**MC6850**
**TABLE 1 — DEFINITION OF ACIA REGISTER CONTENTS**

Data Bus Line Number	Buffer Address			
	RS = R/W	RS = R/W	RS = R/W	RS = R/W
	Transmit Data Register (Write Only)	Receive Data Register (Read Only)	Control Register (Write Only)	Status Register (Read Only)
0	Data Bit 0*	Data Bit 0	Counter Divide Select 1 (CR0)	Receive Data Register Full (RDRF)
1	Data Bit 1	Data Bit 1	Counter Divide Select 2 (CR1)	Transmit Data Register Empty (TORE)
2	Data Bit 2	Data Bit 2	Word Select 1 (CR2)	Data Carrier Detect (DCD)
3	Data Bit 3	Data Bit 3	Word Select 2 (CR3)	Clear to Send (CTS)
4	Data Bit 4	Data Bit 4	Word Select 3 (CR4)	Framing Error (FE)
5	Data Bit 5	Data Bit 5	Transmit Control 1 (CR5)	Receiver Overrun (OVRN)
6	Data Bit 6	Data Bit 6	Transmit Control 2 (CR6)	Parity Error (PE)
7	Data Bit 7***	Data Bit 7**	Receive Interrupt Enable (CR7)	Interrupt Request (IRQ)

\* Leading bit = LSB = Bit 0

\*\* Data bit will be zero in 7 bit plus parity modes.

\*\*\* Data bit is "don't care" in 7 bit plus parity modes.

the data is retained in the RDR. The status is maintained by RDRF as to whether or not the data is current. When the Receive Data Register is full, the automatic transfer of data from the Receiver Shift Register to the Data Register is inhibited and the RDR contents remain valid with its current status stored in the Status Register.

**CONTROL REGISTER**

The ACIA Control Register consists of eight bits of write-only buffer that are selected when RS and R/W are low. This register controls the function of the receiver, transmitter, interrupt enables, and the Request-to-Send peripheral/modem control output.

**Counter Divide Select Bits (CR0 and CR1)** — The Counter Divide Select Bits (CR0 and CR1) determine the divide ratios utilized in both the transmitter and receiver sections of the ACIA. Additionally, these bits are used to provide a master reset for the ACIA which clears the Status Register (except for external conditions on CTS and DCD) and initializes both the receiver and transmitter. Master reset does not affect other Control Register bits. Note that after power-on or a power fail/restart, these bits must be set high to reset the ACIA. After resetting, the clock divide ratio may be selected. These counter select bits provide for the following clock divide ratios:

CR1	CR0	Function
0	0	÷ 1
0	1	÷ 16
1	0	÷ 64
1	1	Master Reset

**Word Select Bits (CR2, CR3, and CR4)** — The Word

Select bits are used to select word length, parity, and the number of stop bits. The encoding format is as follows:

CR4	CR3	CR2	Function
0	0	0	7 Bits + Even Parity + 2 Stop Bits
0	0	1	7 Bits + Odd Parity + 2 Stop Bits
0	1	0	7 Bits + Even Parity + 1 Stop Bit
0	1	1	7 Bits + Odd Parity + 1 Stop Bit
1	0	0	8 Bits + 2 Stop Bits
1	0	1	8 Bits + 1 Stop Bit
1	1	0	8 Bits + Even Parity + 1 Stop Bit
1	1	1	8 Bits + Odd Parity + 1 Stop Bit

Word length, Parity Select, and Stop Bit changes are not buffered and therefore become effective immediately.

**Transmitter Control Bits (CR5 and CR6)** — Two Transmitter Control bits provide for the control of the interrupt from the Transmit Data Register Empty condition, the Request-to-Send (RTS) output, and the transmission of a Break level (space). The following encoding format is used:

CR6	CR5	Function
0	0	RTS = low, Transmitting Interrupt Disabled.
0	1	RTS = low, Transmitting Interrupt Enabled.
1	0	RTS = high, Transmitting Interrupt Disabled.
1	1	RTS = low, Transmits a Break level on the Transmit Data Output. Transmitting Interrupt Disabled.

**Receive Interrupt Enable Bit (CR7)** — The following interrupts will be enabled by a high level in bit position 7 of the Control Register (CR7): Receive Data Register Full, Overrun, or a low to high transition on the Data Carrier Detect (DCD) signal line.


**MOTOROLA Semiconductor Products Inc.**

## MC6850

## STATUS REGISTER

Information on the status of the ACIA is available to the MPU by reading the ACIA Status Register. This read-only register is selected when RS is low and R/W is high. Information stored in this register indicates the status of the Transmit Data Register, the Receive Data Register and error logic, and the peripheral/modem status inputs of the ACIA.

**Receive Data Register Full (RDRF), Bit 0** — Receive Data Register Full indicates that received data has been transferred to the Receive Data Register. RDRF is cleared after an MPU read of the Receive Data Register or by a master reset. The cleared or empty state indicates that the contents of the Receive Data Register are not current. Data Carrier Detect being high also causes RDRF to indicate empty.

**Transmit Data Register Empty (TDRE), Bit 1** — The Transmit Data Register Empty bit being set high indicates that the Transmit Data Register contents have been transferred and that new data may be entered. The low state indicates that the register is full and that transmission of a new character has not begun since the last write data command.

**Data Carrier Detect (DCD), Bit 2** — The Data Carrier Detect bit will be high when the DCD input from a modem has gone high to indicate that a carrier is not present. This bit going high causes an Interrupt Request to be generated when the Receive Interrupt Enable is set. It remains high after the DCD input is returned low until cleared by first reading the Status Register and then the Data Register or until a master reset occurs. If the DCD input remains high after read status and read data or master reset has occurred, the interrupt is cleared, the DCD status bit remains high and will follow the DCD input.

**Clear-to-Send (CTS), Bit 3** — The Clear-to-Send bit indicates the state of the Clear-to-Send input from a modem. A low CTS indicates that there is a Clear-to-Send from the modem. In the high state, the Transmit Data Register Empty bit is inhibited and the Clear-to-Send status bit will be high. Master reset does not affect the

Clear-to-Send Status bit.

**Framing Error (FE), Bit 4** — Framing error indicates that the received character is improperly framed by a start and a stop bit and is detected by the absence of the 1st stop bit. This error indicates a synchronization error, faulty transmission, or a break condition. The framing error flag is set or reset during the receive data transfer time. Therefore, this error indicator is present throughout the time that the associated character is available.

**Receiver Overrun (OVRN), Bit 5** — Overrun is an error flag that indicates that one or more characters in the data stream were lost. That is, a character or a number of characters were received but not read from the Receive Data Register (RDR) prior to subsequent characters being received. The overrun condition begins at the midpoint of the last bit of the second character received in succession without a read of the RDR having occurred. The Overrun does not occur in the Status Register until the valid character prior to Overrun has been read. The RDRF bit remains set until the Overrun is reset. Character synchronization is maintained during the Overrun condition. The Overrun indication is reset after the reading of data from the Receive Data Register or by a Master Reset.

**Parity Error (PE), Bit 6** — The parity error flag indicates that the number of highs (ones) in the character does not agree with the preselected odd or even parity. Odd parity is defined to be when the total number of ones is odd. The parity error indication will be present as long as the data character is in the RDR. If no parity is selected, then both the transmitter parity generator output and the receiver parity check results are inhibited.

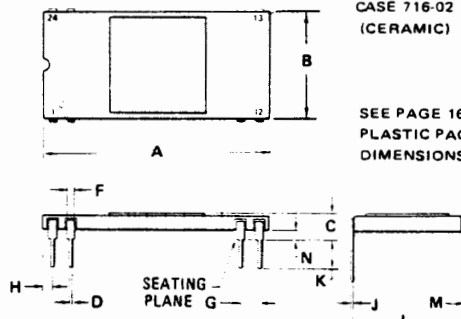
**Interrupt Request (IRQ), Bit 7** — The IRQ bit indicates the state of the IRQ output. Any interrupt condition with its applicable enable will be indicated in this status bit. Anytime the IRQ output is low the IRQ bit will be high to indicate the interrupt or service request status. IRQ is cleared by a read operation to the Receive Data Register or a write operation to the Transmit Data Register.

## PIN ASSIGNMENT

1	VSS	CTS	24
2	Rx Data	DCD	23
3	Rx Clk	D0	22
4	Tx Clk	D1	21
5	RTS	D2	20
6	Tx Data	D3	19
7	IRQ	D4	18
8	CS0	D5	17
9	CS2	D6	16
10	CS1	D7	15
11	RS	E	14
12	VDD	R/W	13

## PACKAGE DIMENSIONS

CASE 716-02  
(CERAMIC)



SEE PAGE 165 FOR  
PLASTIC PACKAGE  
DIMENSIONS

DIM	MILLIMETERS		INCHES	
	MIN	MAX	MIN	MAX
A	29.97	30.99	1.180	1.220
B	14.88	15.62	0.585	0.615
C	3.05	4.19	0.120	0.165
D	0.38	0.53	0.015	0.021
F	0.76	1.40	0.030	0.055
G	2.54 BSC		0.100 BSC	
H	0.76	1.78	0.030	0.070
J	0.20	0.30	0.008	0.012
K	2.54	4.19	0.100	0.165
L	14.88	15.37	0.585	0.605
M	10°		10°	
N	0.51	1.52	0.020	0.060

## NOTE:

1. LEADS TRUE POSITIONED WITHIN 0.25mm (0.010) DIA (AT SEATING PLANE) AT MAXIMUM MATERIAL CONDITION.



MOTOROLA Semiconductor Products Inc.


**MOTOROLA**  
**Semiconductors**

BOX 20912 • PHOENIX, ARIZONA 85036

### 128 X 8-BIT STATIC RANDOM ACCESS MEMORY

The MCM6810 is a byte-organized memory designed for use in bus-organized systems. It is fabricated with N-channel silicon-gate technology. For ease of use, the device operates from a single power supply, has compatibility with TTL and DTL, and needs no clocks or refreshing because of static operation.

The memory is compatible with the M6800 Microcomputer Family, providing random storage in byte increments. Memory expansion is provided through multiple Chip Select inputs.

- Organized as 128 Bytes of 8 Bits
- Static Operation
- Bi-Directional Three-State Data Input/Output
- Six Chip Select Inputs (Four Active Low, Two Active High)
- Single 5-Volt Power Supply
- TTL Compatible
- Maximum Access Time = 350 ns – MCM6810AL 1  
450 ns – MCM6810AL

#### ABSOLUTE MAXIMUM RATINGS (See Note 1)

Rating	Symbol	Value	Unit
Supply Voltage	$V_{CC}$	-0.3 to +7.0	Vdc
Input Voltage	$V_{in}$	-0.3 to +7.0	Vdc
Operating Temperature Range	$T_A$	0 to +70	°C
Storage Temperature Range	$T_{stg}$	-65 to +150	°C

NOTE 1. Permanent device damage may occur if ABSOLUTE MAXIMUM RATINGS are exceeded. Functional operation should be restricted to RECOMMENDED OPERATING CONDITIONS. Exposure to higher than recommended voltages for extended periods of time could affect device reliability.

**MCM6810A**

(0 to 70°C; L or P Suffix)

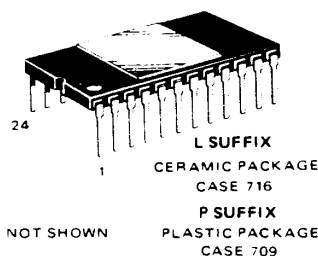
**MCM6810AC**

(-40 to 85°C; L Suffix only)

**MOS**

(N-CHANNEL, SILICON-GATE)

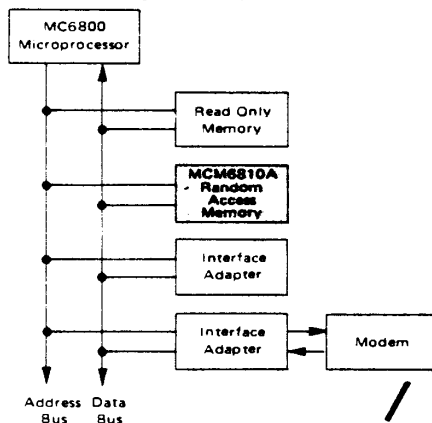
### 128 X 8-BIT STATIC RANDOM ACCESS MEMORY



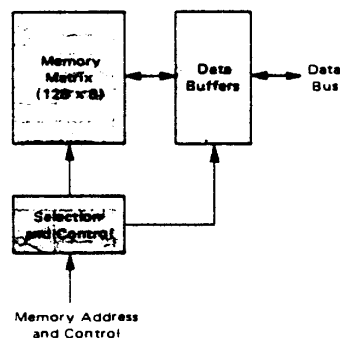
#### PIN ASSIGNMENT

1	Gnd	$V_{CC}$	24
2	D0	A0	23
3	D1	A1	22
4	D2	A2	21
5	D3	A3	20
6	D4	A4	19
7	D5	A5	18
8	D6	A6	17
9	D7	R/W	16
10	CS0	CS5	15
11	CS1	CS4	14
12	CS2	CS3	13

#### M6800 MICROCOMPUTER FAMILY BLOCK DIAGRAM



#### MCM6810A RANDOM ACCESS MEMORY BLOCK DIAGRAM



## MCM6810A

## DC OPERATING CONDITIONS AND CHARACTERISTICS

(Full operating voltage and temperature range unless otherwise noted.)

## RECOMMENDED DC OPERATING CONDITIONS

Parameter	Symbol	Min	Nom	Max	Unit
Supply Voltage	$V_{CC}$	4.75	5.0	5.25	Vdc
Input High Voltage	$V_{IH}$	2.0	—	5.25	Vdc
Input Low Voltage	$V_{IL}$	-0.3	—	0.8	Vdc

## DC CHARACTERISTICS

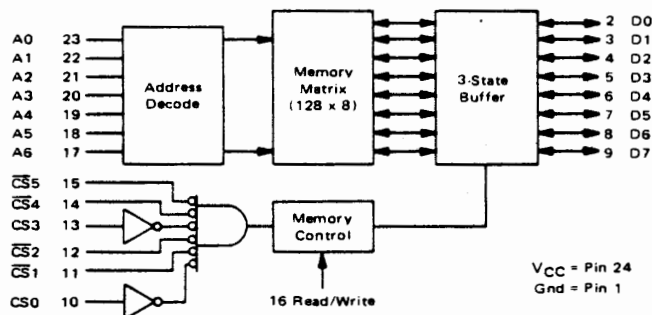
Characteristic	Symbol	Min	Typ	Max	Unit
Input Current ( $I_{A_n}$ , R/W, $CS_n$ , $\overline{CS}_n$ ) ( $V_{in} = 0$ to 5.25 V)	$I_{in}$	—	—	2.5	$\mu$ Adc
Output High Voltage ( $I_{OH} = -205 \mu$ A)	$V_{OH}$	2.4	—	—	Vdc
Output Low Voltage ( $I_{OL} = 1.6$ mA)	$V_{OL}$	—	—	0.4	Vdc
Output Leakage Current (Three-State) ( $CS = 0.8$ V or $\overline{CS} = 2.0$ V, $V_{out} = 0.4$ V to 2.4 V)	$I_{LO}$	—	—	10	$\mu$ Adc
Supply Current ( $V_{CC} = 5.25$ V, all other pins grounded, $T_A = 0^\circ$ C) MCM6810AL MCM6810AL1	$I_{CC}$	—	—	70 80	mAdc

**CAPACITANCE** ( $f = 1.0$  MHz,  $T_A = 25^\circ$ C, periodically sampled rather than 100% tested.)

Characteristic	Symbol	Max	Unit
Input Capacitance	$C_{in}$	7.5	pF
Output Capacitance	$C_{out}$	12.5	pF

This device contains circuitry to protect the inputs against damage due to high static voltages or electric fields; however, it is advised that normal precautions be taken to avoid application of any voltage higher than maximum rated voltages to this high-impedance circuit.

## BLOCK DIAGRAM



MOTOROLA Semiconductor Products Inc.



**MCM6810A**

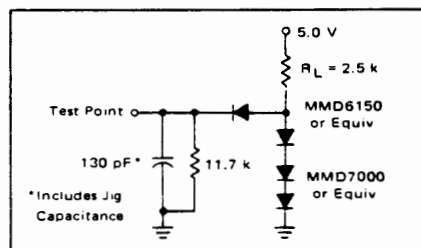
**AC OPERATING CONDITIONS AND CHARACTERISTICS**

(Full operating voltage and temperature unless otherwise noted.)

**AC TEST CONDITIONS**

Condition	Value
Input Pulse Levels	0.8 V to 2.0 V
Input Rise and Fall Times	20 ns
Output Load	See Figure 1

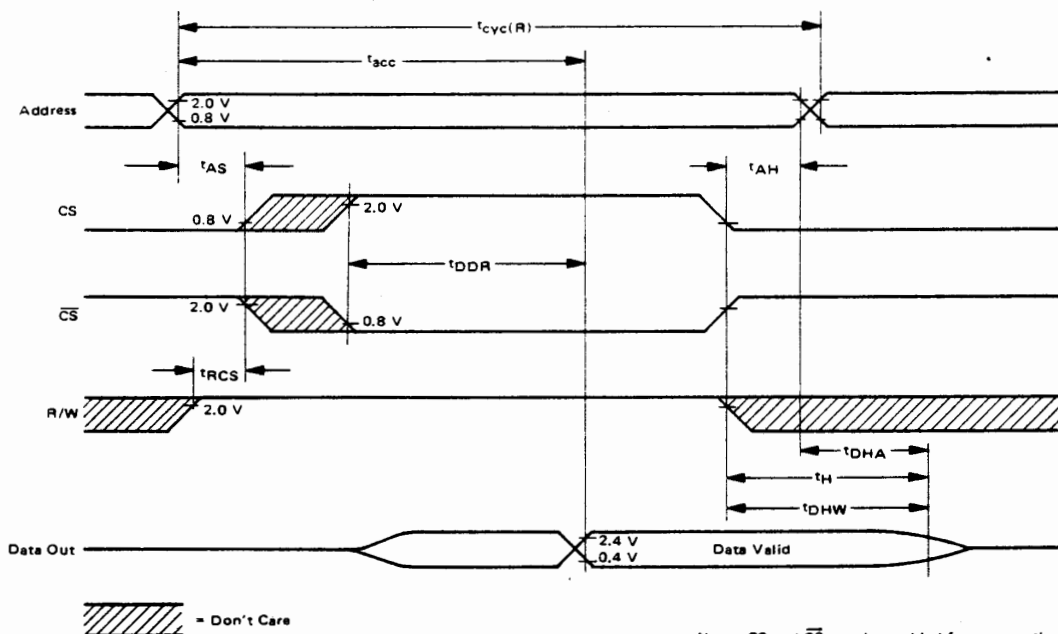
**FIGURE 1 - AC TEST LOAD**



**READ CYCLE**

Characteristic	Symbol	MCM6810AL		MCM6810AL1		Unit
		Min	Max	Min	Max	
Read Cycle Time	$t_{cyc(R)}$	450	—	350	—	ns
Access Time	$t_{acc}$	—	450	—	350	ns
Address Setup Time	$t_{AS}$	20	—	20	—	ns
Address Hold Time	$t_{AH}$	0	—	0	—	ns
Data Delay Time (Read)	$t_{DDR}$	—	230	—	180	ns
Read to Select Delay Time	$t_{RCS}$	0	—	0	—	ns
Data Hold from Address	$t_{DHA}$	10	—	10	—	ns
Output Hold Time	$t_H$	10	—	10	—	ns
Data Hold from Write	$t_{DHW}$	10	80	10	60	ns

**READ CYCLE TIMING**



Note: CS and  $\overline{CS}$  can be enabled for consecutive read cycles provided R/W remains at  $V_{IH}$ .



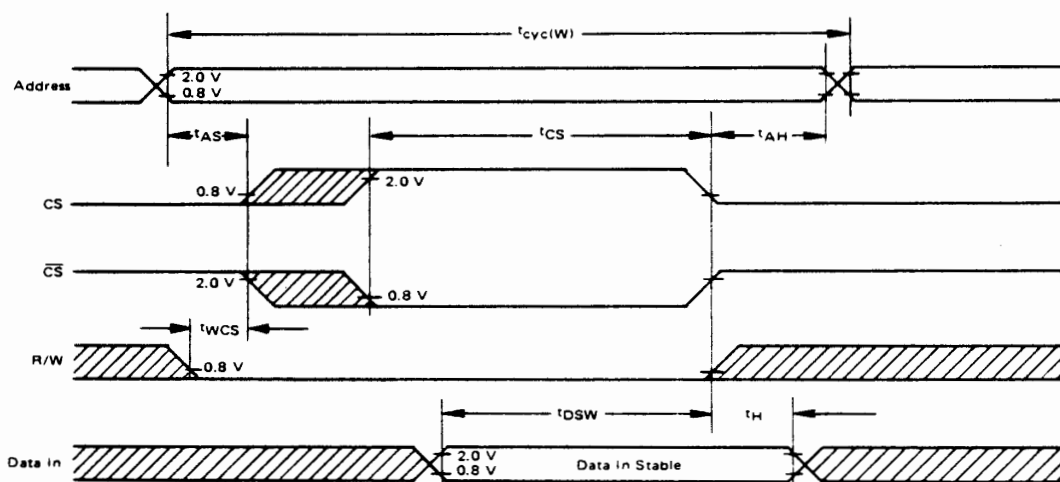
**MOTOROLA Semiconductor Products Inc.**

## MCM6810A

## WRITE CYCLE

Characteristic	Symbol	MCM6810AL		MCM6810AL1		Unit
		Min	Max	Min	Max	
Write Cycle Time	$t_{\text{cyc(W)}}$	450	—	350	—	ns
Address Setup Time	$t_{\text{AS}}$	20	—	20	—	ns
Address Hold Time	$t_{\text{AH}}$	0	—	0	—	ns
Chip Select Pulse Width	$t_{\text{CS}}$	300	—	250	—	ns
Write to Chip Select Delay Time	$t_{\text{WCS}}$	0	—	0	—	ns
Data Setup Time (Write)	$t_{\text{DSW}}$	190	—	150	—	ns
Input Hold Time	$t_{\text{H}}$	10	—	10	—	ns

## WRITE CYCLE TIMING

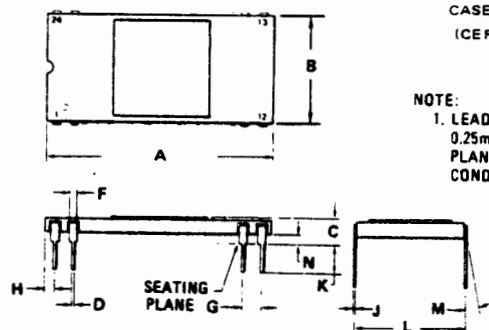


= Don't Care

Note: CS and  $\overline{\text{CS}}$  can be enabled for consecutive write cycles provided R/W is strobed to  $V_{\text{IH}}$  before or coincident with the Address change, and remains high for time  $t_{\text{AS}}$ .

## PACKAGE DIMENSIONS

 CASE 716-02  
 (CERAMIC)

 See Page 165 for  
 Plastic Package dimensions.


## NOTE:

- LEADS TRUE POSITIONED WITHIN 0.25mm (0.010) DIA (AT SEATING PLANE) AT MAXIMUM MATERIAL CONDITION.

DIM	MILLIMETERS		INCHES	
	MIN	MAX	MIN	MAX
A	29.97	30.99	1.180	1.220
B	14.88	15.62	0.585	0.615
C	3.05	4.19	0.120	0.165
D	0.38	0.53	0.015	0.021
F	0.76	1.40	0.030	0.055
G	2.54 BSC		0.100 BSC	
H	0.76	1.78	0.030	0.070
J	0.20	0.30	0.008	0.012
K	2.54	4.19	0.100	0.165
L	14.88	15.37	0.585	0.605
M	—	$10^0$	—	$10^0$
N	0.51	1.52	0.020	0.060


**MOTOROLA Semiconductor Products Inc.**


**MOTOROLA**  
**Semiconductors**

BOX 20912 • PHOENIX, ARIZONA 85036

## Advance Information

### 1024 X 8-BIT READ ONLY MEMORY

The MCM6830A is a mask-programmable byte-organized memory designed for use in bus-organized systems. It is fabricated with N-channel silicon-gate technology. For ease of use, the device operates from a single power supply, has compatibility with TTL and DTL, and needs no clocks or refreshing because of static operation.

The memory is compatible with the M6800 Microcomputer Family, providing read only storage in byte increments. Memory expansion is provided through multiple Chip Select inputs. The active level of the Chip Select inputs and the memory content are defined by the customer.

- Organized as 1024 Bytes of 8 Bits
- Static Operation
- Three-State Data Output
- Four Chip Select Inputs (Programmable)
- Single 5-Volt Power Supply
- TTL Compatible
- Maximum Access Time = 500 ns

### ABSOLUTE MAXIMUM RATINGS (See Note 1)

Rating	Symbol	Value	Unit
Supply Voltage	$V_{CC}$	-0.3 to +7.0	Vdc
Input Voltage	$V_{in}$	-0.3 to +7.0	Vdc
Operating Temperature Range	$T_A$	0 to +70	°C
Storage Temperature Range	$T_{stg}$	-65 to +150	°C

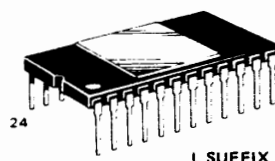
NOTE 1 Permanent device damage may occur if ABSOLUTE MAXIMUM RATINGS are exceeded. Functional operation should be restricted to RECOMMENDED OPERATING CONDITIONS. Exposure to higher than recommended voltages for extended periods of time could affect device reliability.

## MCM6830A

### MOS

(N-CHANNEL, SILICON-GATE)

### 1024 X 8-BIT READ ONLY MEMORY

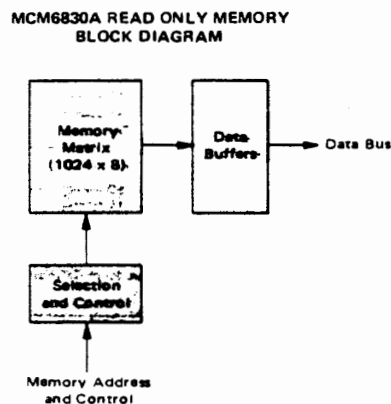
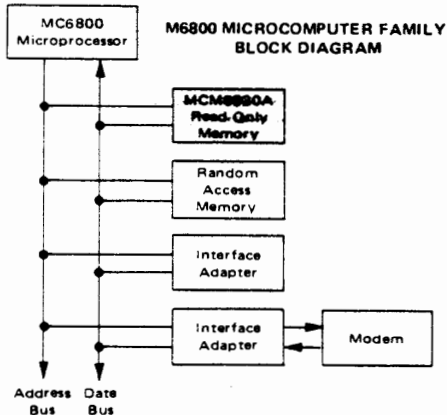


**L SUFFIX**  
 CERAMIC PACKAGE  
 CASE 716

**P SUFFIX**  
 PLASTIC PACKAGE  
 CASE 709

### PIN ASSIGNMENT

1	Gnd	A0	24
2	D0	A1	23
3	D1	A2	22
4	D2	A3	21
5	D3	A4	20
6	D4	A5	19
7	D5	A6	18
8	D6	A7	17
9	D7	A8	16
10	CS0	A9	15
11	CS1	CS3	14
12	VCC	CS2	13



This is advance information and specifications are subject to change without notice.

## MCM6830A

## DC OPERATING CONDITIONS AND CHARACTERISTICS

(Full operating voltage and temperature range unless otherwise noted.)

## RECOMMENDED DC OPERATING CONDITIONS

Parameter	Symbol	Min	Nom	Max	Unit
Supply Voltage	$V_{CC}$	4.75	5.0	5.25	Vdc
Input High Voltage	$V_{IH}$	2.0	—	5.25	Vdc
Input Low Voltage	$V_{IL}$	-0.3	—	0.8	Vdc

## DC CHARACTERISTICS

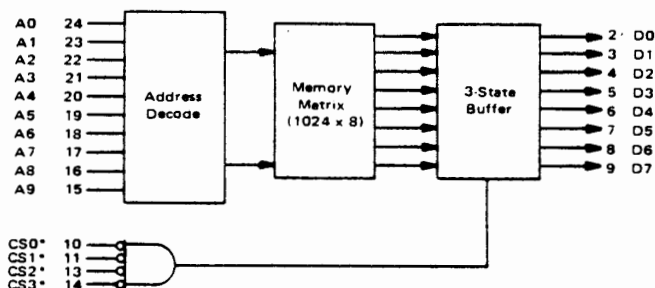
Characteristic	Symbol	Min	Typ	Max	Unit
Input Current ( $V_{in} = 0$ to 5.25 V)	$I_{in}$	—	—	2.5	$\mu A_{dc}$
Output High Voltage ( $I_{OH} = -205 \mu A$ )	$V_{OH}$	2.4	—	—	Vdc
Output Low Voltage ( $I_{OL} = 1.6 \text{ mA}$ )	$V_{OL}$	—	—	0.4	Vdc
Output Leakage Current (Three-State) ( $CS = 0.8 \text{ V}$ or $\overline{CS} = 2.0 \text{ V}$ , $V_{out} = 0.4 \text{ V}$ to $2.4 \text{ V}$ )	$I_{LO}$	—	—	10	$\mu A_{dc}$
Supply Current ( $V_{CC} = 5.25 \text{ V}$ , $T_A = 0^\circ C$ )	$I_{CC}$	—	—	130	$\text{mA}_{dc}$

CAPACITANCE ( $f = 1.0 \text{ MHz}$ ,  $T_A = 25^\circ C$ , periodically sampled rather than 100% tested.)

Characteristic	Symbol	Max	Unit
Input Capacitance	$C_{in}$	7.5	pF
Output Capacitance	$C_{out}$	12.5	pF

This device contains circuitry to protect the inputs against damage due to high static voltages or electric fields; however, it is advised that normal precautions be taken to avoid application of any voltage higher than maximum rated voltages to this high-impedance circuit.

## BLOCK DIAGRAM



\* Active level defined by the customer.

 $V_{CC}$  = Pin 12  
 Gnd = Pin 1


MOTOROLA Semiconductor Products Inc.

**MCM6830A**

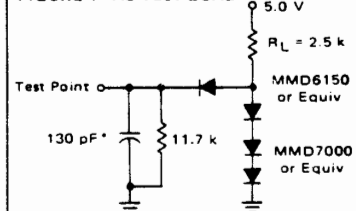
**AC OPERATING CONDITIONS AND CHARACTERISTICS**

(Full operating voltage and temperature unless otherwise noted.)

(All timing with  $t_r = t_f = 20$  ns, Load of Figure 1)

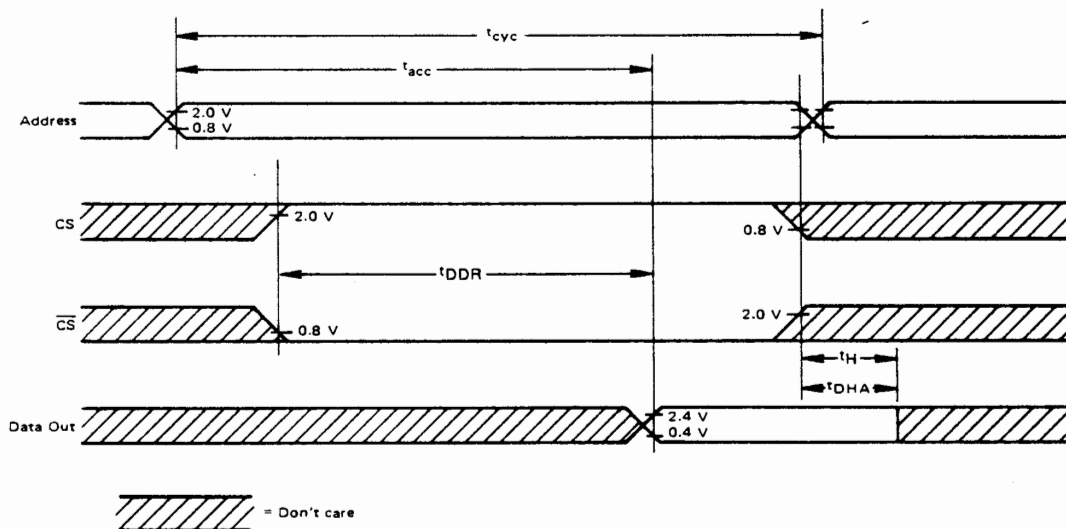
Characteristic	Symbol	Min	Max	Unit
Cycle Time	$t_{cyc}$	500	—	ns
Access Time	$t_{acc}$	—	500	ns
Data Delay Time (Read)	$t_{DDR}$	—	300	ns
Data Hold from Address	$t_{DHA}$	10	—	ns
Data Hold from Deselection	$t_H$	10	150	ns

**FIGURE 1—AC TEST LOAD**



\*Includes Jig Capacitance

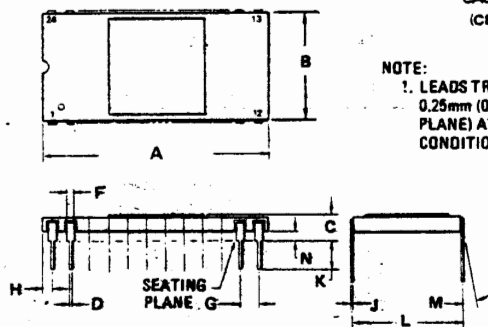
**TIMING DIAGRAM**



**PACKAGE DIMENSIONS**

CASE 716-02  
(CERAMIC)

NOTE:  
1. LEADS TRUE POSITIONED WITHIN  
0.25mm (0.010) DIA (AT SEATING  
PLANE) AT MAXIMUM MATERIAL  
CONDITION.



See Page 185 for  
Plastic Package dimensions.

DIM	MILLIMETERS		INCHES	
	MIN	MAX	MIN	MAX
A	29.97	30.99	1.180	1.220
B	14.88	15.62	0.585	0.615
C	3.05	4.19	0.120	0.165
D	0.38	0.53	0.015	0.021
F	0.76	1.40	0.030	0.055
G	2.54 BSC		0.100 BSC	
H	0.76	1.78	0.030	0.070
J	0.20	0.30	0.008	0.012
K	2.54	4.19	0.100	0.165
L	14.88	15.37	0.585	0.605
M	—	10 <sup>0</sup>	—	10 <sup>0</sup>
N	0.51	1.52	0.020	0.060



**MOTOROLA Semiconductor Products Inc.**

**MCM6830A**
**CUSTOM PROGRAMMING**

By the programming of a single photomask for the MCM6830A, the customer may specify the content of the memory and the method of enabling the outputs.

Information on the general options of the MCM6830A should be submitted on an Organizational Data form such as that shown in Figure 3.

Information for custom memory content may be sent to Motorola in one of two forms (shown in order of preference):

1. Paper tape output of the Motorola M6800 Software.
2. Hexadecimal coding using IBM Punch Cards.

**PAPER TAPE**

Included in the software packages developed for the M6800 Microcomputer Family is the ability to produce a paper tape output for computerized mask generation. The assembler directives are used to control allocation of memory, to assign values for stored data, and for controlling the assembly process. The paper tape must specify the full 1024 bytes.

Note: Motorola can accept magnetic tape and truth table formats. For further information, contact your local Motorola sales representative.

**FIGURE 2 - BINARY TO HEXADECIMAL CONVERSION**

Binary Data				Hexadecimal Character
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	8
1	0	0	1	9
1	0	1	0	A
1	0	1	1	B
1	1	0	0	C
1	1	0	1	D
1	1	1	0	E
1	1	1	1	F

**IBM PUNCH CARDS**

The hexadecimal equivalent (from Figure 2) may be placed on 80 column IBM punch cards as follows:

Step	Column	
1	12	Byte "0" Hexadecimal equivalent for outputs D7 thru D4 (D7 = M.S.B.)
2	13	Byte "0" Hexadecimal equivalent for outputs D3 thru D0 (D3 = M.S.B.)
3	14-75	Alternate steps 1 and 2 for consecutive bytes.
4	77-78	Card number (starting 01)
5	79-80	Total number of cards (32)

**FIGURE 3 - FORMAT FOR PROGRAMMING GENERAL OPTIONS**

<b>ORGANIZATIONAL DATA</b> <b>MCM6830A MOS READ ONLY MEMORY</b>																				
<div style="display: flex; justify-content: space-between;"> <div style="width: 60%;"> <p>Customer:</p> <p>Company _____</p> <p>Part No. _____</p> <p>Originator _____</p> <p>Phone No. _____</p> </div> <div style="width: 35%; border: 1px solid black; padding: 5px;"> <p style="text-align: center;">Motorola Use Only:</p> <p>Quote: _____</p> <p>Part No.: _____</p> <p>Specif. No.: _____</p> </div> </div>																				
<p>Enable Options:</p> <table style="width: 100%; margin-top: 10px;"> <tr> <td></td> <td style="text-align: center;">1</td> <td style="text-align: center;">0</td> <td></td> </tr> <tr> <td style="text-align: right;">CS0</td> <td style="text-align: center;"><input type="checkbox"/></td> <td style="text-align: center;"><input type="checkbox"/></td> <td rowspan="4" style="vertical-align: top; padding-left: 20px;">           1 is most positive            0 is most negative         </td> </tr> <tr> <td style="text-align: right;">CS1</td> <td style="text-align: center;"><input type="checkbox"/></td> <td style="text-align: center;"><input type="checkbox"/></td> </tr> <tr> <td style="text-align: right;">CS2</td> <td style="text-align: center;"><input type="checkbox"/></td> <td style="text-align: center;"><input type="checkbox"/></td> </tr> <tr> <td style="text-align: right;">CS3</td> <td style="text-align: center;"><input type="checkbox"/></td> <td style="text-align: center;"><input type="checkbox"/></td> </tr> </table>					1	0		CS0	<input type="checkbox"/>	<input type="checkbox"/>	1 is most positive 0 is most negative	CS1	<input type="checkbox"/>	<input type="checkbox"/>	CS2	<input type="checkbox"/>	<input type="checkbox"/>	CS3	<input type="checkbox"/>	<input type="checkbox"/>
	1	0																		
CS0	<input type="checkbox"/>	<input type="checkbox"/>	1 is most positive 0 is most negative																	
CS1	<input type="checkbox"/>	<input type="checkbox"/>																		
CS2	<input type="checkbox"/>	<input type="checkbox"/>																		
CS3	<input type="checkbox"/>	<input type="checkbox"/>																		


**MOTOROLA Semiconductor Products Inc.**


**MOTOROLA**  
**Semiconductors**

BOX 20912 • PHOENIX, ARIZONA 85036

## Advance Information

### 2048 x 8-BIT READ ONLY MEMORY

The MCM6832 is a mask-programmable byte-organized memory designed for use in bus-organized systems. It is fabricated with N-channel metal-gate technology. For ease of use, the device is compatible with TTL and DTL, and needs no clocks or refreshing because of static operation.

The memory is compatible with the M6800 Microcomputer Family, providing read only storage in byte increments. Memory expansion is provided through a Chip Select input. The active level of the Chip Select input and the memory content are defined by the customer.

- Organized as 2048 Bytes of 8 Bits
- Static Operation
- Three-State Data Output
- Programmable Chip Select
- TTL Compatible
- Maximum Access Time = 500 ns

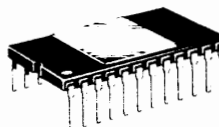
### ABSOLUTE MAXIMUM RATINGS<sup>1</sup> (Referenced to V<sub>SS</sub>)

Rating	Symbol	Value	Unit
Supply Voltages	V <sub>DD</sub>	-0.3 to +15	Vdc
	V <sub>CC</sub>	-0.3 to +6.0	
	V <sub>BB</sub>	-10 to +0.3	
Address/Control Input Voltage	V <sub>in</sub>	-0.3 to +15	Vdc
Operating Temperature Range	T <sub>A</sub>	0 to +70	°C
Storage Temperature Range	T <sub>stg</sub>	-55 to +125	°C

Note 1: Permanent device damage may occur if ABSOLUTE MAXIMUM RATINGS are exceeded. Functional operation should be restricted to RECOMMENDED OPERATING CONDITIONS. Exposure to higher than recommended voltages for extended periods of time could affect device reliability.

**MCM6832**
**MOS**

(N-CHANNEL, LOW THRESHOLD)

**2048 x 8-BIT  
READ ONLY MEMORY**

**L SUFFIX**

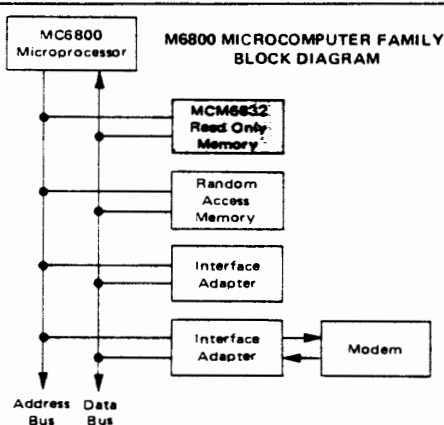
 CERAMIC PACKAGE  
CASE 716

NOT SHOWN: P SUFFIX

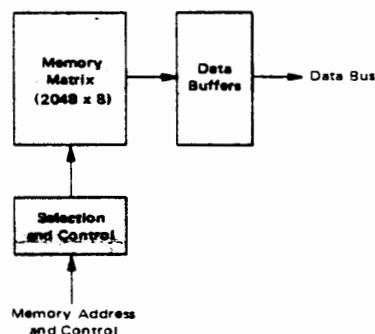
 PLASTIC PACKAGE  
CASE 709

### PIN ASSIGNMENT

1	V <sub>BB</sub>	V <sub>CC</sub>	24
2	A <sub>10</sub>	V <sub>DD</sub>	23
3	CS	A <sub>9</sub>	22
4	D <sub>0</sub>	A <sub>8</sub>	21
5	D <sub>1</sub>	A <sub>7</sub>	20
6	D <sub>2</sub>	D <sub>4</sub>	19
7	D <sub>3</sub>	D <sub>5</sub>	18
8	A <sub>0</sub>	D <sub>6</sub>	17
9	A <sub>1</sub>	D <sub>7</sub>	16
10	A <sub>2</sub>	A <sub>6</sub>	15
11	A <sub>3</sub>	A <sub>5</sub>	14
12	V <sub>SS</sub>	A <sub>4</sub>	13



### MCM6832 READ ONLY MEMORY BLOCK DIAGRAM



This is advance information and specifications are subject to change without notice.

## MCM6832

**DC OPERATING CONDITIONS AND CHARACTERISTICS**  
 (Full operating voltage and temperature range unless otherwise noted.)

**RECOMMENDED DC OPERATING CONDITIONS** (Referenced to  $V_{SS}$  = Ground)

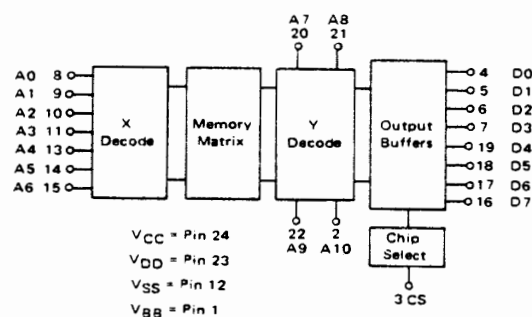
Parameter	Symbol	Min	Typ	Max	Unit
Supply Voltage	$V_{DD}$	11.4	12	12.6	Vdc
	$V_{CC}$	4.75	5.0	5.25	Vdc
	$V_{BB}$	-5.25	-5.0	-4.75	Vdc
Input High Voltage ( $A_n$ , CS)	$V_{IH}$	3.0	—	$V_{CC}$	Vdc
Input Low Voltage ( $A_n$ , CS)	$V_{IL}$	-0.3	—	0.8	Vdc

**DC CHARACTERISTICS**

Characteristic	Symbol	Min	Typ	Max	Unit
Input Leakage Current ( $A_n$ , CS) ( $V_{in} = 0$ to 5.25 V)	$I_{in}$	—	—	10	$\mu$ Adc
Output Leakage Current (Three-State) ( $V_O = 0.4$ V to -2.4 V, CS = 0.4 V or CS = 2.4 V.)	$I_{LO}$	—	—	10	$\mu$ Adc
Output High Voltage ( $I_{OH} = -100 \mu$ A)	$V_{OH}$	3.7	—	$V_{CC}$	Vdc
Output Low Voltage ( $I_{OL} = 1.6$ mA)	$V_{OL}$	0	—	0.4	Vdc
Supply Current (Chip Deselected or Selected)	$I_{DD}$	—	—	25	mAdc
	$I_{CC}$	—	—	45	mAdc
	$I_{BB}$	—	—	500	$\mu$ Adc

**CAPACITANCE** (Periodically Sampled Rather Than 100% Tested.)

Characteristic	Symbol	Min	Typ	Max	Unit
Input Capacitance ( $f = 1$ MHz)	$C_{in}$	—	5.0	7.5	pF
Output Capacitance ( $f = 1$ MHz)	$C_{out}$	—	5.0	10	pF

**BLOCK DIAGRAM**

**MOTOROLA Semiconductor Products Inc.**



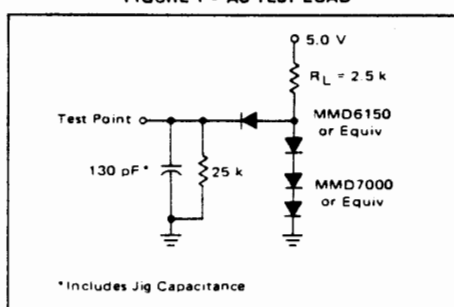
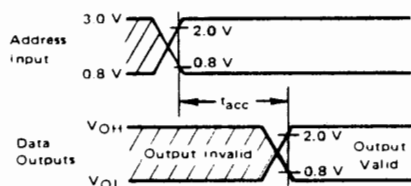
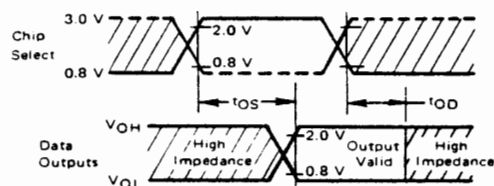
MCM6832

**AC CHARACTERISTICS**

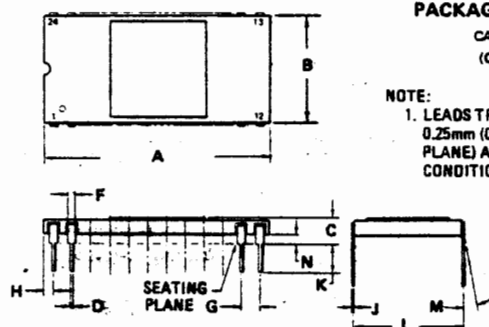
 (Full operating voltage and temperature unless otherwise noted. All timing with  $t_r = t_f \leq 20$  ns;  
 Load = 1 TTL Gate (MC7400 Series) biased to draw 1.6 mA;  $C_L = 130$  pF.)

Characteristic	Symbol	Min	Typ*	Max	Unit
Address Access Time	$t_{acc}$	—	320*	500	ns
Output Select Time	$t_{OS}$	—	175*	300	ns
Output Deselect Time	$t_{OD}$	30	100*	150	ns

\*Typical values measured at 25°C and nominal supply voltages.

**FIGURE 1 — AC TEST LOAD**

**FIGURE 2 — TIMING DIAGRAM**
**A ADDRESS ACCESS TIMING DIAGRAM**  
 (Chip Selected)

**B CHIP SELECT TIMING DIAGRAM**  
 (Addresses Established)

**PACKAGE DIMENSIONS**

 CASE 716-02  
 (CERAMIC)

 NOTE:  
 1. LEADS TRUE POSITION WITHIN  
 0.25mm (0.010) DIA (AT SEATING  
 PLANE) AT MAXIMUM MATERIAL  
 CONDITION.

 SEE PAGE 168 FOR  
 PLASTIC PACKAGE  
 DIMENSIONS.

DIM	MILLIMETERS		INCHES	
	MIN	MAX	MIN	MAX
A	29.97	30.99	1.180	1.220
B	14.88	15.62	0.585	0.615
C	3.05	4.19	0.120	0.165
D	0.38	0.53	0.015	0.021
F	0.76	1.40	0.030	0.055
G	2.54 BSC		0.100 BSC	
H	0.76	1.78	0.030	0.070
J	0.20	0.30	0.008	0.012
K	2.54	4.19	0.100	0.165
L	14.88	15.37	0.585	0.605
M	—	10 <sup>0</sup>	—	10 <sup>0</sup>
N	0.51	1.52	0.020	0.060


**MOTOROLA Semiconductor Products Inc.**

## MCM6832

## CUSTOM PROGRAMMING

By the programming of a single photomask for the MCM6832, the customer may specify the content of the memory and the method of enabling the outputs.

Information on the general options of the MCM6832 should be submitted on an Organizational Data form such as that shown in Figure 4.

Information for custom memory content may be sent to Motorola in one of two forms (shown in order of preference):

1. Paper tape output of the Motorola M6800 Software.
2. Hexadecimal coding using IBM Punch Cards.

## PAPER TAPE

Included in the software packages developed for the M6800 Microcomputer Family is the ability to produce a paper tape output for computerized mask generation. The assembler directives are used to control allocation of memory, to assign values for stored data, and for controlling the assembly process. The paper tape must specify the full 2048 bytes.

Note: Motorola can accept magnetic tape and truth table formats. For further information, contact your local Motorola sales representative.

FIGURE 3 - BINARY TO HEXADECIMAL CONVERSION

MSB D7 D3	D6 D2	D5 D1	LSB D4 D0	Hexadecimal Character
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	8
1	0	0	1	9
1	0	1	0	A
1	0	1	1	B
1	1	0	0	C
1	1	0	1	D
1	1	1	0	E
1	1	1	1	F

0 = VOL  
1 = VOH

## IBM PUNCH CARDS

The hexadecimal equivalent (from Figure 3) may be placed on 80 column IBM punch cards as follows:

Step	Column	
1	12	Byte "0" Hexadecimal equivalent for outputs D7 thru D4 (D7 = M.S.B.)
2	13	Byte "0" Hexadecimal equivalent for outputs D3 thru D0 (D3 = M.S.B.)
3	14-75	Alternate steps 1 and 2 for consecutive bytes.
4	77-78	Card number (starting 01)
5	79-80	Total number of cards (64)

FIGURE 4 - FORMAT FOR PROGRAMMING GENERAL OPTIONS

ORGANIZATIONAL DATA MCM6832 MOS READ ONLY MEMORY	
Customer: <div style="display: flex; justify-content: space-between; margin-top: 10px;"> <div style="width: 60%;">             Company _____              Part No. _____              Originator _____              Phone No. _____           </div> <div style="width: 35%; border: 1px solid black; padding: 5px; margin-top: 10px;">             Motorola Use Only              Quote _____              Part No. _____              Specif. No. _____           </div> </div> <div style="margin-top: 20px;">             True Chip Select Options:             <div style="display: flex; justify-content: space-around; margin-left: 100px;"> <div style="text-align: center;">               I.    1    <input type="checkbox"/> </div> <div style="text-align: center;">               II.   0    <input type="checkbox"/> </div> </div> <div style="text-align: right; margin-right: 50px; margin-top: 10px;">               1 is most positive                0 is most negative             </div> </div>	


**MOTOROLA Semiconductor Products Inc.**

## POSITIVE POWERS OF 2

n	$2^n$	
0	1	
1	2	
2	4	
3	8	
4	16	
5	32	
6	64	
7	128	
8	256	
9	512	
10	1024	
11	2048	
12	4096	
13	8192	
14	16384	
15	32768	
16	65536	
17	131072	
18	262144	
19	524288	
20	1048576	
21	2097152	
22	4194304	
23	8388608	
24	16777216	
25	33554432	
26	67108864	
27	134217728	
28	268435456	
29	536870912	
30	1073741824	
31	2147483648	
32	4294967296	

## NEGATIVE POWERS OF 2

n	$2^{-n}$					
0	1.0					
1	0.5					
2	0.25					
3	0.125					
4	0.0625					
5	0.03125					
6	0.01562	5				
7	0.00781	25				
8	0.00390	625				
9	0.00195	3125				
10	0.00097	65625				
11	0.00048	82812	5			
12	0.00024	41406	25			
13	0.00012	20703	125			
14	0.00006	10351	5625			
15	0.00003	05175	78125			
16	0.00001	52587	89062	5		
17	0.00000	76293	94531	25		
18	0.00000	38146	97265	625		
19	0.00000	19073	48632	8125		
20	0.00000	09536	74316	40625		
21	0.00000	04768	37158	20312	5	
22	0.00000	02384	18579	10156	25	
23	0.00000	01192	09289	55078	125	
24	0.00000	00596	04644	77539	0625	
25	0.00000	00298	02322	38769	53125	
26	0.00000	00149	01161	19384	76562	5
27	0.00000	00074	50580	59692	38281	25
28	0.00000	00037	25290	29846	19140	625
29	0.00000	00018	62645	14923	09570	3125
30	0.00000	00009	31322	57461	54785	15625
31	0.00000	00004	65661	28730	77392	57812 5
32	0.00000	00002	32830	64365	38696	28906 25

## POSITIVE POWERS OF 8

n	8 <sup>n</sup>
0	1
1	8
2	64
3	512
4	4096
5	32768
6	262144
7	2097152
8	16777216

## POSITIVE POWERS OF 16

n	16 <sup>n</sup>
0	1
1	16
2	256
3	4096
4	65536
5	1048576
6	16777216
7	268435456
8	4294967296

## NEGATIVE POWERS OF 16

n	16 <sup>-n</sup>
0	1.0
1	0.0625
2	0.00390625
3	0.000244140625
4	0.0000152587890625

